A Multi-Colony Ant Algorithm for Optimizing Join Queries in Distributed Database Systems


Faculty of Electrical and Computer Engineering Shahid Beheshti University, G.C., Tehran, Iran
ladan.golshanara@gmail.com, rohani@sbu.ac.ir, tasom2002@yahoo.com

Received: Jan 07, 2012 / Revised: Jul 12, 2012 / Accepted: Nov 11, 2012

Abstract. Distributed database systems provide a new data processing and storage technology for decentralized organizations of today. Query optimization, the process to generate an optimal execution plan for the posed query, is more challenging in such systems due to the huge search space of alternative plans incurred by distribution. As finding an optimal execution plan is computationally intractable, using stochastic-based algorithms has drawn the attention of most researchers. In this paper, for the first time, a multi-colony ant algorithm is proposed for optimizing join queries in a distributed environment where relations can be replicated but not fragmented. In the proposed algorithm, four types of ants collaborate to create an execution plan. Hence, there are four ant colonies in each iteration. Each type of ant makes an important decision to find the optimal plan. In order to evaluate the quality of the generated plan, two cost models are used- one based on the total time and the other on the response time. The proposed algorithm is compared with two previous genetic-based algorithms on chain, tree and cyclic queries. The experimental results show that the proposed algorithm saves up to about 80% of optimization time with no significant difference in the quality of generated plans compared with the best existing genetic-based algorithm.

Keywords: Query optimization; distributed database system; ant colony optimization; join queries; cost model.
1. Introduction

Decentralized nature of today’s organizations and enterprises along with their massive volume of data demands for new data processing and data keeping technologies. Distributed database systems seem to meet these needs while providing more reliability, availability and parallelism compared to the conventional centralized database systems.

The performance of distributed database systems highly relies on two interrelated processes: (1) its design, i.e. how to fragment, replicate and allocate data, so that frequent queries run as locally as possible, and its method of query optimization. In fact, as user’s queries are expressed at a high semantic level, query optimization is known to be a big challenge in both centralized and distributed relational database systems (Date 2004). Query optimization in different environments- from single processor relational database systems to data grid systems- is discussed in (Hameurlain 2009). Optimization gets harder in distributed database systems because such systems should provide distribution, network and performance transparencies so that the users feel as if they are working with a centralized database system. Good surveys of distributed query optimization are provided in (Aljanaby et al. 2005; Kossmann 2000; Yu et al. 1984).

Queries can be categorized according to various criteria such as:

- The type of operation, which includes retrieve and update queries,
- The number of tuples retrieved, which includes range and singleton queries,
- The type of operator used, which includes set, aggregate and join queries,
- The application type, which includes Online Transaction Processing (OLTP) and Decision Support (DSS) queries,
- …

This paper only considers the optimization of join queries, also known as Select-Project-Join (SPJ) queries, because they are not only more common, but their optimization also makes a significant contribution to the overall efficiency of database systems. By optimization, we mean cost-based optimization in which a cost model is used to predict the cost of execution plans. There are other kinds of optimizations such as semantic optimization which takes advantage of business rules -i.e. integrity constraints- to simplify queries (Date 2004), multiple query
optimization} which tries to find same patterns in submitted queries and obtain a
global plan for their optimization (Ioannidis 1996), and \textit{rule-based optimization}
which optimizes queries based on a set of rules ranked in the order of efficiency
(Connolly et al. 2005). The problem of optimizing multiple-join queries has been
proved to be computationally intractable with a large number of relations (Ibaraki
et al. 1984).

There are three phases of distributed join queries optimization (Rho et al. 1997;
Yu et al. 1984): \textit{copy identification} phase, \textit{reduction} phase and \textit{assembly} phase.
\textit{The copy identification} phase identifies the appropriate replica(s) of relations
referenced by the query. The goal of \textit{reduction} phase is to reduce the cardinalities
of relations involved before transmitting them so that the communication costs are
minimized. Thus, in this phase, reduction operators such as required selections,
projections and semi-joins are applied. Finally, in the \textit{assembly} phase, join
operations are performed at join sites and required transmissions are done.

Generally speaking, for optimization problems, a strategy should be devised to
explore the search space of possible solutions and find the optimal solution based
on their objective function values. The search space for query optimization
consists of several equivalent query execution plans (QEPs)- obtained by
employing transformation rules such as commutativity and associativity of the
join operator- which differ in performance (Ozsu et al. 2011). Unlike the typical
optimizers of centralized environments that only consider linear QEPs, both linear
and bushy QEPs should be taken into account in distributed environments to
exploit the potential parallelism these environments offer. Therefore, the search
space becomes larger especially when relations are replicated. Coping with this
huge search space needs powerful search strategy algorithms. The objective
function of a cost-based distributed query optimizer can be based on either the
response time (i.e. the elapsed time from the initiation to the completion of the
query), the total time (i.e. the sum of all local processing time and communication
time) or a (weighted) combination of both. The choice between these cost models
depends on the application program. For example, while for DSS queries
minimizing the response time is preferred, for OLTP queries, it is minimizing the
total time which is preferred (Gorla 2010).

In this paper, an ant colony optimization (ACO) algorithm is introduced for the
first time to tackle the problem of distributed join query optimization in a search
space where relations are replicated, and not fragmented. Ant algorithms had been previously applied for designing distributed database systems (i.e. fragmentation and allocation) in (Goli et al. 2011; Karimi Adl et al. 2009). Furthermore, ant and bee colony algorithms have been applied for join query optimization in centralized database systems which have led to better results compared to genetic algorithms (Alamery et al. 2010; Li et al. 2008). Two cost models are introduced—one based on the total time, and the other on the response time. Our cost models consider both communication as well as local processing costs. We have compared our algorithm to two other genetic algorithms previously proposed in (Rho et al. 1997; Sevinç et al. 2011), the results of which show that our proposed algorithm could find reasonable QEPs with a little optimization overhead.

The rest of this paper is structured as follows: Section 2 covers related previous work, basic definitions are introduced in Section 3, and Section 4 states the problem of distributed join query optimization. Section 5 explains some general concepts of ACO algorithms which our proposed algorithm is based on. The search space, cost models and search strategy of our proposed algorithm are depicted in Section 6, while the steps of the proposed MMAS-DJQO algorithm are described in Section 7. Section 8 describes two alternative algorithms which our proposed algorithm is compared with, and Section 9 presents the experimental results. Finally, Section 10 discusses the conclusion and future work.

2. Related work

Due to its importance, there is a rich literature on query optimization. In a paper like this, it would be impossible to cover all the proposed algorithms; therefore, we only mention some of the ideas behind those closely related to our proposed algorithm. Algorithms used in the first distributed database systems such as R*, SDD-1 and Distributed-Ingres are widely studied and compared in (Ozsu et al. 2011); thus, there is no need to repeat them here.

In 1980s and early 1990s, when communication costs were the dominant factor in cost models, a lot of research had been done on reduction of operands in order to minimize the amount of data transmitted over the network. It is important to mention these strategies since the proposed algorithm uses one of them, and could be easily extended to consider other techniques as well. Some of these strategies are as follows:
• Semi-joins (Bernstein et al. 1981a) are applied to reduce one of the relations involved in a join through sending join attribute(s) of one of the relations to the site of the other so as to extract matching tuples.

• Two-way semi-joins (2SJ) (Roussopoulos et al. 1991) are the extension of semi-joins to reduce both relations involved in a join.

• Bit vector filter (Babb 1979) further reduces transmission costs by sending a bit vector instead of the join attribute which contains the same joining information.

• PERF join (Li et al. 1995) is designed to reduce the cost of 2SJ backward phase.

One way to classify distributed query optimization algorithms depends on whether they take advantage of fragmentation or not. The algorithms in the first category that take advantage of (horizontal) fragmentation- such as Fragment and Replicate Strategy (FRS) (Yu et al. 1987) or Partition and Replicate Strategy (PRS) (Yu et al. 1989) - enable parallel processing of a query. However, there are some restrictions on these algorithms. For example, the FRS cannot be applied if none of the relations referenced by the query is fragmented, and in the PRS, as the number of relations increases, the percentage of improvement decreases (Yu et al. 1989). The proposed algorithm in this paper falls into the second category, i.e. it does not consider fragmentation. Hence, we focus on the recent previous work in this area, which use metaheuristic techniques.

In 1997, Ribero and his colleagues proposed an algorithm with the following characteristics (Ribiero et al. 1997):

• Its search space contains both bushy and linear execution plans. Unlike other algorithms that usually avoid unnecessary Cartesian products, Ribero et al. argue that Cartesian products may be worth doing in distributed environments, in case the relations are located at the same site. Their proposed algorithm does not benefit from semi-joins and does the optimization in an environment where no relations are replicated.

• The cost model is based on the response time in terms of pipelining of join operations. The relevant original ideas came from cost models which are used for parallel databases (Lanzelotte et al 1994), though may not be appropriate to be applied in distributed databases1.

• The search strategy is based on Tabu search metaheuristic algorithm and it uses a hashing-based data structure for Tabu search memory.
In the same year, Rho and his colleague presented a comprehensive distributed query optimization model that integrates copy identification, join order, join site selection and reduction by semi-joins into a single model (Rho et al. 1997). Their proposed algorithm, which is called OGA97 throughout this paper, has the following features:

- Its search space contains execution plans that reflects an environment where not only can relations be replicated, but reduction with semi-joins are also considered and joins can be executed at different sites rather the result site.
- The cost model is based on the total cost; both local processing and communication costs are considered. The cost of transferring the results to the result site is also included in the cost model.
- OGA97 algorithm uses a genetic-based search strategy.

In 2011, another genetic-based optimization algorithm was proposed by Sevinc et al. (Sevinç et al. 2011), called NGA11 algorithm in this paper. Its search space is similar to that of OGA97, except that relations cannot be replicated. Its cost model is based on the total cost.

The algorithms which perform optimization based on a cost model before executing the query (i.e. in compile time)- like the above algorithms- are static optimization algorithms. In contrast, dynamic optimization algorithms, such as (Shahabi et al. 2009; Yu et al. 1997), do optimization and execution at the same time. In fact, the dynamic query optimization algorithm, introduced in (Yu et al. 1997), works in distributed Internet databases where too much uncertainty prevents estimating the cost of a QEP. Also, the algorithm proposed in (Shahabi et al. 2009) is for dynamic distributed database systems, where values are frequently changed.

To our knowledge, no swarm intelligence algorithm has been so far proposed as a search strategy for distributed join query optimization. However, these algorithms are successfully applied in centralized query optimization (Alamery et al. 2010; Li et al. 2008) and distributed database design (Goli et al. 2011; Karimi Adl et al. 2009). Our proposed algorithm employs the model once introduced in OGA97 algorithm and its search strategy is based on the ant colony optimization metaheuristic. It is compared to OGA97 and NGA11 algorithms, which, as far as

---

1 Pipelining operations in distributed database systems may not be as worthy as parallel database systems; because, in distributed environments, every time a tuple is ready to be sent, a TCP connection has to be
we know, are the only algorithms in the literature analogous to ours. Other algorithms confine their search space to one of the decisions the distributed optimizer has to make. For example, the Tabu search algorithm discussed above and the algorithms surveyed in (Aljanaby et al. 2005) restrict the distributed query optimization problem to a search for the best execution order, the proposed algorithm in (Gorla et al. 2010) narrows down the problem to find the best sites to allocate join operations and the strategies proposed in (Babb 1979; Bernstein et al. 1981a; Roussopoulos et al. 1991; Li et al. 1995) focus on reduction caused by semi-joins. Therefore, they are not comparable with our algorithm which its optimization model integrates all these decisions. In contrast to OGA97 and NGA11 algorithms, we use two cost models- one based on the total time and the other based on the response time.

3. Preliminaries

In this section we introduce basic definitions which will be used throughout the paper.

**Definition 1. Join Queries:** A join query is one very typical kind of queries that have at least one join operation. Join queries can be shown by means of *join graphs*. The vertices of the join graph represent the relations involved in the join query and the edges show join attributes. Depending on the structure of the join graph, join queries can be categorized into cyclic and acyclic (or tree) queries. Cyclic queries have a cycle in their join graphs and tree queries have a join graph with a tree structure. Tree queries can be classified into simple, chain and star queries. All edges of simple queries contain only one join attribute. The vertices of a chain query have at most two edges, each belonging to a different join attribute. Star queries, as the name suggests, have a join graph with one central relation that is joined with *n* other relations. Hence, there are *n* join attributes in the central relation (Unnava 1992).

**Definition 2. Semi-join:** Semi-join operator can act as a reducer of operand relations. The idea is to reduce the communication cost by first sending the join attribute(s) of one relation to the site of the other and then sending all matching tuples from the second relation back to the site of the first relation for final join operation (Bonabeau et al. 1999). The relation which sends its join attribute is established which is quite time consuming.
called the reducer, and the reduced relation is called the reduce (Rho et al. 1997).
The semi-join will be beneficial if the cost of its production and transfer to the
other site is less than the cost of the transfer of the whole operand relation. SDD-1
optimizer (Bernstein et al. 1981b), which was designed for slow wide area
networks, makes extensive use of semi-joins.

Semi-joins which deal with two base (not intermediate) relations are called
elementary semi-joins. Either of base relations could be the reducer, hence
reducing the other relation. Therefore, there are two possible elementary semi-
joins for each join operation (Rho et al. 1997). In this paper, we show semi-joins
by an arrow from the reducer relation to the reduced relation, \( R_i \rightarrow R_j \).

Gainful semi-joins (Chen et al. 1993) do not result in reducing the current join, but
contribute to reducing the cost of subsequent join operations, and as a result,
reduce the overall cost of executing a query.

**Definition 3. Query Execution Plan (QEP):** A query execution plan includes
information about executing the submitted query. QEPs are typically abstracted
by means of annotated query trees. If the only operation considered is join
operation, the query tree is called join tree. The leaves of the join tree represent
base relations, the intermediate nodes represent join operations and the root
represents the final step. In the join trees of Fig. 1, the root vertex of each tree is
labeled \( R \), the intermediate nodes are labeled \( J_i \) (where \( i \) is the number of join
operation) and the leaves are labeled \( R_i \) (\( i \) is the number of relation). With respect
to their shape, two types of join trees could be distinguished: linear and bushy. In
linear trees, the relation on one side of each join operator is always a base relation
(Connolly et al. 2005). By contrast, in bushy trees, both operands may be
intermediate relations which make this type useful in illustrating parallelism
(Ozsu et al. 2011). The typical optimizers of centralized database systems restrict
the search space to left-deep trees – a kind of linear join tree in which the right
operand of each join operation is a base relation– allowing them to take advantage
of dynamic processing techniques. However, this reduction of search space leads
to neglect of many alternative execution strategies and hence, is not acceptable in
distributed environments.

Further information on the environment can be added by annotating the join tree
(Ozsu et al. 2011; Rho et al. 1997). In Fig. 1, the root of each tree is annotated
with the query result site. The intermediate nodes are annotated with the sites at
which join operations are performed and the leaves are annotated with the sites
from which relations are accessed. For instance, in the bushy tree of Fig. 1(a), join
$J_3$ is performed at site $S_3$ and relation $R_3$ is accessed from site $S_2$, while in the
annotated left-deep tree of Fig. 1(b), join $J_3$ is performed at site $S_2$ and relation $R_3$
is accessed from site $S_i$. Semi-joins are represented by a directed arrow from the
reducing relation to the reduced one (Rho et al. 1997). For example in Fig. 1(a),
relation $R_2$ sends its join attribute(s) to the sites containing relations $R_1$ and $R_3$, as
reduced relations. On the other hand, in the tree in Fig. 1(b), relations $R_1$ and $R_3$
are reducing relations and relations $R_2$ and $R_d$ are the reduced ones.

![Diagram](image_url)

**Fig. 1** Examples of two annotated join trees for a query with three joins: (a) A bushy tree (b) A
linear tree

**Definition 4.** Pure join attribute: Pure join attributes (Chen et al. 1993) are join
attributes which do not appear in the final result of a query. Therefore, these join
attributes can be eliminated as soon as the corresponding join operations are
executed (i.e. they need not to be transmitted with the result of the join).

**Definition 5.** Cost model: A cost model provides a way to estimate the cost of a
QEP. In distributed databases, the cost model could be based on either the total
time or the response time. The total time is the sum of all time (cost) components,
while the response time is the interval between the submission of a query and the
arrival of the complete response. A general formula for the total time is (Ozsu et
al. 2011):

$$Total\_time = T_{cpu} \times \#\text{insts} + T_{I/O} \times \#I/Os + T_{MSG} \times \#msgs + T_{IR} \times \#bytes$$
The two first components depict local processing time, where \( T_{cpu} \) is the time of a CPU instruction, \#insts the number of CPU instructions, \( T_{I/O} \) the time of a disk I/O and \#I/Os is the number of input/output operations. Communication time is measured by the two last component- \( T_{MSG} \) is the fixed time of initiating and receiving a message, \#msgs the number of messages, \( T_{TR} \) the time it takes to transmit a data unit from one site to another and \#bytes is the number of bytes transmitted. Minimizing the total time entails the utilization of resources (such as CPUs, I/Os and communication channels); thus, more queries can be processed in a given time i.e. the system throughput increases (Gorla et al. 2010).

A general formula for the response time is (Ozsu et al. 2011):

\[
\text{Response \_ time} = T_{cpu} \times \text{seq \_ #insts} + T_{I/O} \times \text{seq \_ #I/Os} + T_{MSG} \times \text{seq \_ #msgs} + T_{TR} \times \text{seq \_ #bytes}
\]

where seq \_ #x , in which x can be instructions ( insts ), I/Os , messages ( msgs ) or bytes , is the maximum number of x which must be done sequentially for the execution of the query. Minimizing the response time is obtained by increasing the degree of parallelism that requires higher resource consumption and leads to reduction in system throughput (Gorla 2010).

As the cost is usually expressed in terms of time units, in the previous literature, as well as this paper, time and cost are used interchangeably.

**Definition 6. Independent parallelism:** Independent parallelism is a case of inter-operator parallelism; the other case is pipeline parallelism. Independent parallelism is achieved when there is no dependency between the operators that are executed in parallel. Bushy query trees are the only query trees that allow this form of parallelism (Ozsu et al. 2011); hence, they are vastly used in distributed environments.

4. Problem Statement

**Distributed Join Query Optimization (DJQO)**

In distributed databases, data are distributed based on a distribution-allocation scheme and may also be replicated at different sites. In such systems, the submitted query may need data which are stored at different sites- i.e. a global
query. A query optimizer aims to find an execution strategy to execute a query that can minimize an objective cost function which could be based on either the total time or the response time. The strategy which is usually represented by a query plan, determines the suitable replica(s) of each relation, the order of join operations, the join sites where the joins are performed and a semi-join program for each join. Finding the best strategy for executing a query is computationally intractable and -due to the combinatorial search space- the real goal of the optimizer is to find a near optimum strategy and more importantly, to avoid bad strategies (Ozsu et al. 2011).

In short, in order to propose an algorithm for query optimization, three components should be addressed: (1) search space as a set of equivalent QEPs to abstract execution strategies, (2) cost model to give a cost to each QEP in order to show the quality (i.e. performance) of each plan and (3) search strategy to investigate the search space in order to find an optimum plan. It is important that the optimizer makes a reasonable compromise between the time to select a QEP and its quality.

In this paper, which is motivated by good results of applying ant algorithms in distributed database design and centralized query optimization, an ant-based algorithm is examined as a search strategy of the distributed query optimizer to see in what circumstances it works better than existing genetic-based algorithms. We also introduce a response time cost model which exploits independent parallelism.

5. General concepts of ACO algorithms

Ant colony optimization (ACO) algorithms are a series of ant algorithms inspired by foraging behavior of ant colonies. To solve a problem with ACO algorithms, first the problem should be abstracted as a graph. For example, the traveling salesman problem (TSP) can be represented by a graph $G = (N, E)$ where a fixed set of vertices $N$ represents the cities and a fixed set of edges $E$ shows the connection between the cities; the objective is to find Hamiltonian path for $G$ which gives the minimal length (Dorigo et al. 2004). Ants build their solutions by moving on the problem graph and laying pheromone so as to guide other ants. Pheromone trails, which provide a positive feedback mechanism, permit ants to cooperate and exploit each other’s experiences. A negative feedback mechanism
is also necessary to avoid stagnation (i.e. premature convergence) which is implemented through pheromone evaporation. This indirect communication mediated by the environment is called stigmergy. Pheromone deposition and evaporation are two phases of pheromone update that happen in the ACO.

Ant System (AS)- the ancestor of all ACO algorithms- was the first ACO algorithm proposed by Dorigo et al. (Dorigo et al. 1996). One of its extensions on which our proposed algorithm is based, is Max-Min Ant System (MMAS) (St et al. 2000). The MMAS algorithm works as follows: pheromone values on all edges of the problem graph are initially set to some fixed value $\tau_0$. This value is recommended to be slightly higher than the expected amount of pheromone deposited by the ants in one iteration; a rough estimate of this value (for MMAS algorithm) is $1/(\rho * C^{\text{nm}})$ where $\rho$ is pheromone evaporation rate and $C^{\text{nm}}$ the cost of solution produced by nearest-neighbor heuristic (i.e. greedy solution) (Dorigo et al. 2004). In each of the iterations, a set of $m$ ants is located at randomly selected vertices of the problem graph with each starting to construct a feasible solution. During solution construction, ants are guided by pheromone and heuristic information as well. Pheromone information reflects the experience acquired by ants during the execution of the algorithm. Heuristic information, on the other hand, indicates the heuristic desirability (i.e. visibility) of moving from a current node to another in the problem graph. For example, in the travelling salesman problem, each ant has to find the best possible tour (i.e. a permutation of cities). The heuristic desirability of choosing city $j$ when in city $i$, is in inverse proportion to the distance between the two cities- the shorter the distance, the higher is the heuristic desirability. After building the solutions, the pheromone trails are updated either by the best-so-far ant or the iteration-best ant; this update reinforces good paths and helps the ants converge to a solution. In the end, the best solution found by this algorithm is reported.

Stagnation occurs when all the ants follow the same path and construct the same solution. To avoid this, the MAX-MIN algorithm limits the pheromone trails to an interval $[\tau_{\text{min}}, \tau_{\text{max}}]$. The values of the upper and lower bounds must be determined experimentally based on the problem at hand. Moreover, if stagnation is detected while iterations proceed, all pheromone trails are re-initialized to $\tau_{\text{max}}$ (Dorigo et al. 2004).
Unlike the original MMAS algorithm, our algorithm uses four colonies of ants in each of the iterations. In fact, in our algorithm, a solution (i.e. a QEP) is created by four types of ants, each in charge of one of the decisions made by the distributed query optimizer to find the optimum execution strategy. Moreover, before outputting the best solution, this algorithm takes advantage of a local search step so as to adjust the best solution by searching its adjacent space. Our algorithm, called MMAS-DJQO\(^2\), is explained in detail in sections 6 and 7.

6. Proposed MMAS algorithm for distributed join query optimization

In this section, the proposed algorithm is described based on its search space, cost model and search strategy.

6.1. Search Space

The search space of our algorithm is similar to the OGA97 algorithm (Rho et al. 1997) for comparison reasons. The main features of our search space are:

- both bushy and linear QEPs are considered,
- The database could be replicated (i.e. a relation may be stored at multiple sites),
- Elementary semi-joins and gainful semi-joins are considered.

6.2. Cost Model

The cost model of OGA97 and NGA11 algorithms is based on the total time. In this paper we use two cost models, one based on the total time and the other on the response time so as to exploit the potential parallelism introduced in distributed environments. In our cost models, the I/O cost represents local processing costs and the transfer cost is the only communication cost considered; therefore, not only CPU costs but also message initiation costs are ignored. Table 1 summarizes the key notations used in this section.

\(^2\) Max-Min Ant System-Distributed Join Query Optimization
### Table 1 Notation description

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_i$</td>
<td>The $i$th relation</td>
</tr>
<tr>
<td>$S_i$</td>
<td>The $i$th site</td>
</tr>
<tr>
<td>$R_i'$</td>
<td>The $i$th relation after being reduced by semi-join</td>
</tr>
<tr>
<td>$J_i$</td>
<td>The $i$th join</td>
</tr>
<tr>
<td>$A_w$</td>
<td>Join attribute</td>
</tr>
<tr>
<td>$\text{card}(R_i)$</td>
<td>The cardinality of relation $R_i$ (The number of tuples $R_i$)</td>
</tr>
<tr>
<td>$\text{SF}_{sj}(R_i,A_w)$</td>
<td>The semi-join selectivity for join attribute $A_w$ in relation $R_i$</td>
</tr>
<tr>
<td>$V(R_i,A_w)$</td>
<td>The number of distinct values of $A_w$ in relation $R_i$</td>
</tr>
<tr>
<td>$\text{card(dom}(A_w)))$</td>
<td>The number of values in attribute domain of $A_w$</td>
</tr>
<tr>
<td>$R_i \text{ join}_{A_w} R_j$</td>
<td>The join of $R_i$ and $R_j$ on join attribute $A_w$</td>
</tr>
<tr>
<td>$R_i \rightarrow R_j$</td>
<td>A semi-join where $R_i$ is the reducer</td>
</tr>
<tr>
<td>$\text{len}(A_w)$</td>
<td>The average length of attribute $A_w$ (byte)</td>
</tr>
<tr>
<td>$\text{len}(R_i)$</td>
<td>The average length of relation $R_i$</td>
</tr>
<tr>
<td>$\text{SF}_{sj}(R_i,A_w)$</td>
<td>Selectivity factor of attribute $A_w$ of relation $R_i$</td>
</tr>
<tr>
<td>$ps$</td>
<td>Page Size</td>
</tr>
<tr>
<td>$P_{R_i}$</td>
<td>The number of page accesses for reading relation $R_i$</td>
</tr>
<tr>
<td>$P_{\text{join}}$</td>
<td>The number of page accesses for join operation</td>
</tr>
<tr>
<td>$P_{\text{write}}$</td>
<td>The number of page accesses for writing intermediate relation on disk</td>
</tr>
<tr>
<td>$\text{IO}_{\text{proj}}(R_i,S_k)$</td>
<td>The I/O time for projecting join attribute of relation $R_i$ at site $S_k$</td>
</tr>
<tr>
<td>$\text{IO}_{sj}$</td>
<td>The I/O time of semi-join operation</td>
</tr>
<tr>
<td>$\text{IO}(S_i)$</td>
<td>The disk I/O time in site $S_i$</td>
</tr>
<tr>
<td>$\text{COM}_{R_i}$</td>
<td>Communication time for transferring relation $R_i$ to another site</td>
</tr>
<tr>
<td>$\text{COM}(S_i,S_j)$</td>
<td>The time to transfer one byte from one site to another</td>
</tr>
<tr>
<td>left ($u$) (or right ($u$))</td>
<td>Left (or right) relation (base or intermediate) of vertex $u$</td>
</tr>
<tr>
<td>$RT_u$</td>
<td>The response time of the non-leaf vertex $u$ (i.e. the intermediate relation represented by vertex $u$)</td>
</tr>
</tbody>
</table>

#### 6.2.1. Total Time

The total time is the sum of all the cost components (i.e. all local processing and communication costs), the estimation of which is to be discussed in detail in this section.
Cost of Semi-joins

As stated earlier, semi-joins are used to reduce cardinalities of relations in the reduction phase. Semi-join costs consist of the cost of projecting the join attribute(s) of the first relation, sending them to the site of the second relation and executing the semi-join right there. Assuming a page as the basic unit of disk I/O, the cost of projecting the join attribute of relation $R_i$ in site $S_k$ is the cost of reading the relation into main memory in order to remove unnecessary attributes:

$$IO_{proj}(R_i, S_k) = \left[ \frac{\text{card}(R_i) \cdot \text{len}(R_i)}{ps} \right] \cdot IO(S_k) \quad (1)$$

The cost of transferring the join attribute of relation $R_i$ from site $S_k$ to site $S_p$ is equal to:

$$COM_{A_w} = V(R_i \cdot A_w) \cdot \text{len}(A_w) \cdot COM(S_k, S_p) \quad (2)$$

The cardinality of $R_j$ after reduction by semi-join is (The reduced relation is shown by $R_j'$):

$$\text{card}(R_j') = \text{card}(R_j) \cdot SF_{sj}(R_i, A_w) \quad (3)$$

where $SF_{sj}(R_i, A_w) = \frac{V(R_i, A_w)}{\text{card(dom}(A_w)))}$. In addition to cardinality, the number of distinct values of join attribute is also reduced by semi-join:

$$V(R_j' \cdot A_w) = V(R_j \cdot A_w) \cdot SF_{sj}(R_i, A_w) \quad (4)$$

The number of page accesses required for semi-join is (in terms of block nested loop join):

$$P_{sj} = P_{A_w} + (P_{A_w} \cdot P_{R_j}) \quad (5)$$

$$P_{A_w} = \left[ \frac{V(R_i, A_w) \cdot \text{len}(A_w)}{ps} \right]$$

$$P_{R_j} = \left[ \frac{\text{card}(R_j) \cdot \text{len}(R_j)}{ps} \right]$$

The number of pages needed to write the result to disk is:
\[ P_{\text{write}} = \left\lfloor \frac{\text{card} (R_j) \times \text{len}(R_j)}{ps} \right\rfloor \]  

(6)

Therefore, the cost of performing a semi-join and storing its result in site \( S_p \) is:

\[ IO_{sj} = (P_{sj} + P_{\text{write}}) \times IO(S_p) \]  

(7)

**Cost of joins**

In order to perform the joins, relations (either base or intermediate) should be transmitted to the join sites if necessary. The cost of transferring relation \( R_i \) from site \( S_k \) to site \( S_p \) is:

\[ \text{COM}_{R_i} = \text{card} (R_i) \times \text{len}(R_i) \times \text{COM} (S_k, S_p) \]  

(8)

The number of page accesses required to perform the join operation is (assuming that \( R_j \) is the smaller relation and the join algorithm is block nested loop join\(^3\)):

\[ P_{\text{join}} = P_{R_i} \times P_{R_j}. \]

The number of pages needed to store the results is:

\[ P_{\text{write}} = \left\lfloor \frac{\text{card} (R_i \ \text{join} \ R_j) \times \text{len}(R_i \ \text{join} \ R_j)}{ps} \right\rfloor \]  

(9)

The cardinality of the intermediate relation produced by join is (Garcia-Molina et al. 2002):

\[ \text{card} (R_i \ \text{join}_{A_w} R_j) = \frac{\text{card} (R_i) \times \text{card} (R_j)}{\max (V (R_i \cdot A_w), V (R_j \cdot A_w))} \]  

(10)

If \( A_w \) is a pure join attribute, it could be removed from the heading of the intermediate relation. Hence, the length of each tuple in intermediate relation is: \( \text{len}(R_i) + \text{len}(R_j) - 2 \times \text{len}(A_w) \). Otherwise, if it is a natural join, the length is: \( \text{len}(R_i) + \text{len}(R_j) - \text{len}(A_w) \) and if it is equi-join, the heading of intermediate relation is the union of the headings of \( R_i \) and \( R_j \).

\(^3\) Our proposed ACO algorithm does not depend on the join algorithm. Therefore, other join algorithms can be applied as well.
Therefore, the local processing cost to perform a join at site $S_k$ is:

$$ IO_{\text{join}} = (P_{\text{join}} + P_{\text{write}}) \times IO(S_k) \quad (11) $$

### 6.2.2. Response Time

When the objective function is based on the response time, parallel execution of operations should be considered. In this paper, only independent parallelism is exploited. Hence, in our cost model, a relation is sent as a whole and not on a tuple-by-tuple basis. In Fig. 2, $J_1$ and $J_2$ and the communication time for transferring the results of these joins to site $S_1$ can be done in parallel because these joins are performed at different sites without using the same resources.

![Fig. 2 An example of independent parallelism](image)

In reduction phase, all semi-joins can be done in parallel, except the following case: if the reducers of two semi-joins are stored at one site, and the reducees are held at another, these semi-joins are done sequentially. For example if $R_i$ and $R_j$ - as reducers of semi-joins $R_i \rightarrow R_k$ and $R_j \rightarrow R_z$ - are stored at site $S_m$ and the reducees (i.e. $R_k$ and $R_z$) at a second site as $S_n$, these semi-joins are performed sequentially. The semi-joins in the join trees in Fig. 1(a) and Fig 1(b) can be done in parallel.

After the operation in reduction phase, the query execution plan is evaluated bottom-up. The response time of the intermediate vertex $u$, which represents join $J_i$ at site $S_k$, depends on the site of the previous operations. If previous joins were executed at the same site, the response time of $u$ is:

$$ RT_u = RT_{\text{left}}(u) + COM_{\text{left}}(u) + RT_{\text{right}}(u) + COM_{\text{right}}(u) + IO_{\text{join}} \quad (12) $$

According to this formula, the response time of vertex $u$, is the sum of the response times of previous joins, the communication time to send the results to site $S_k$, and the local processing time for performing the join $J_i$. If the site for previous operations is the same as that for the current join, then
\[ COM_{left}(u) = COM_{right}(u) = 0 \text{ and if the left (right) vertex of the current join is a base relation (i.e. a leaf vertex), then } RT_{left}(u) = 0 \ (RT_{right}(u) = 0). \]

If the previous joins are performed at different sites, the join operations and the transfer of the results can be done in parallel, hence the response time of vertex \( u \) is:

\[ RT_u = \max (RT_{left}(u) + COM_{left}(u), RT_{right}(u) + COM_{right}(u)) + IO_{join} \quad (13) \]

As mentioned before, QEP is evaluated bottom-up. After calculating the response time of the root vertex, it is summed with the maximum cost of semi-joins. The communication time to send the final result to the result site is also added if necessary.

Notice that if there is no semi-join program for a join operation, the join operation can be done in parallel with semi-joins; however, as this makes the cost model too complicated, we do not consider it in our model.

### 6.3. Search strategy

The search strategy for the proposed algorithm is based on ACO metaheuristic. As mentioned before, our proposed algorithm in particular, is based on Max-Min Ant System (MMAS), and hence we call it MMAS-DJQO.

In this algorithm, a different type of ant is used for each of the four decisions that should be made in order to find an optimum strategy. In other words, similar to real life where different ants are specialized to do different tasks (Bonabeau et al. 1999), in our proposed algorithm, different types of ants are responsible for each decision made to find optimum QEP. These ants are described below:

- **JoinAnt** determines the order of join operations. The number of vertices of the graph this ant builds is equal to the number of joins in the query and each vertex represents a unique join operation.
- **SiteAnt** determines the join sites. The number of vertices of the graph this ant builds is equal to the number of joins in the query and each vertex represents the site assigned to a join to execute in.
- **ReplicaAnt** determines the replica for each relation. The number of vertices of the graph this ant builds is equal to the number of relations involved in the submitted join query and each vertex represents the site in which the selected replica is held.
- **SemiAnt** determines the type of semi-join to be used for each join operation. The type of semi-join is selected from the set \{00,01,10\} where '00' means the use of
no semi-join program, '01' means the second relation acts as the reducer (i.e. the one that sends its join attribute to the site of the other relation) and '10' means the first relation is the reducer.

A solution (QEP) is produced after each of these ants builds its graph; the cost of generated QEP is then evaluated based on one of our cost models. The lower the cost, the better the solution is.

*For example*, consider the following query:

```
SELECT *
FROM R1, R2, R3, R4
WHERE R1.A1 = R2.A1
AND R2.A2 = R3.A2
AND R3.A3 = R4.A3
```

This query has three join conditions: $J_1 := R1.A1 = R2.A1$, $J_2 := R2.A2 = R3.A2$ and $J_3 := R3.A3 = R4.A3$. Suppose $R_1$, $R_3$ and $R_4$ are replicated at sites $\{S_2, S_3\}$, $\{S_1, S_2\}$ and $\{S_6, S_3, S_6\}$ respectively; and $R_2$ is held at site $S_7$. Two possible solutions the ants could find are illustrated in Fig. 3. The first solution corresponds to the bushy tree in Fig. 1(a) and the second solution corresponds to the left-deep tree in Fig. 1(b).

In the first solution, JoinAnt finds the permutation $J_1$, $J_3$ and $J_2$ which means these joins should be executed in the same order they are listed. According to the graph of SiteAnt, join $J_1$ will be executed at site $S_2$, while joins $J_2$ and $J_3$ will be performed at site $S_3$. The ReplicaAnt graph depicts the sites where the chosen replicas of $R_1$, $R_2$, $R_3$ and $R_4$ reside. The graph of SemiAnt shows that joins $J_1$ and $J_2$ use a semi-join program while $J_3$ uses no semi-joins (‘00’). In $J_1$ semi-join program, i.e ‘01’, the right relation ($R_2$) is the reducer; however, in $J_2$ semi-join program, i.e. ‘10’, the left relation (again $R_2$) is the reducer. The second solution can be interpreted the same way: the join order is $J_1$, $J_2$, $J_3$, which are respectively executed in sites $S_3$, $S_1$ and $S_2$. According to the semiAnt graph, in both $J_1$ and $J_3$ joins, the left relations- $R_1$ in join $J_1$ and $R_3$ in join $J_3$- are the reducers, while join $J_2$ uses no semi-join program (‘00’).

As soon as all types of ants construct their graphs, a solution (QEP) is produced, which can be attributed a cost based on our cost models. Notice that Solution1 in Fig. 3 has less response time due to the fact that joins $J_1$ and $J_3$ could be done in parallel.
In the above query, since all the attributes are retrieved (owing to SELECT *), there is no pure join attribute; but, if it were ‘SELECT $R_1.A_1, R_2.A_2, R_3.A_3,$’ then, because $A_3$ is not needed in the final result, it would be a pure join attribute.

<table>
<thead>
<tr>
<th>Solution1:</th>
<th>Solution2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>JoinAnt $J_1 \rightarrow J_2 \rightarrow J_3$</td>
<td>SiteAnt $S_2 \rightarrow S_1 \rightarrow S_3$</td>
</tr>
<tr>
<td>ReplicaAnt $S_2 \rightarrow S_1 \rightarrow S_3$</td>
<td>SemiAnt $10 \rightarrow 00 \rightarrow 10$</td>
</tr>
<tr>
<td>$R_1$</td>
<td>$R_2$</td>
</tr>
<tr>
<td>$\text{Cost of Solution1:}$</td>
<td>$\text{Cost of Solution2:}$</td>
</tr>
<tr>
<td>$\text{TotalTime: 0.5sec}$</td>
<td>$\text{TotalTime: 0.4sec}$</td>
</tr>
<tr>
<td>$\text{ResponseTime: 0.25sec}$</td>
<td>$\text{ResponseTime: 0.35sec}$</td>
</tr>
</tbody>
</table>

Fig. 3 Examples of two solutions constructed by our algorithm and their costs

7. Description of MMAS-DJQO

The pseudo-code for the MMAS-DJQO is shown below:

1. Set the algorithm parameters and initialize all pheromone trails to $T_0$.
2. For iter = 1 to MAX-ITERATION
3. For ant = 1 to ANTS-NO
4. JoinAnt, ReplicaAnt, SiteAnt and SemiAnt probabilistically construct a feasible solution according to pheromone information as well as heuristic information.
5. End For
6. Calculate the cost of each solution according to the cost model
7. If the iteration-best solution has lower cost than best-so-far,
8. Update the best-so-far solution
9. End If
10. If there is no improvement in the best-so-far solution for RESET-COUNT consecutive iterations
11. Reinitialize all pheromone trails to $T_{max}$
12. Else
13. Pheromone Evaporation
14. Update pheromone trails based on best-so-far solution
15. Check pheromone not to exceed the pheromone boundaries.
16. End If
17. End For
18. Improve the best-so-far solution with local search
19. Output the best-so-far solution
7.1. Initialization

In the first step, the algorithm parameters (e.g. maximum number of iterations, pheromone evaporation rate, etc.) are set and all pheromone trails are initialized to \( \tau_0 \) which is equal to \( 1/(\rho * C^{GR}) \), where \( C^{GR} \) is the cost of solution found by greedy algorithm and \( \rho \) is the pheromone evaporation rate. The greedy solution is calculated as follows: join order is determined by sorting joins in increasing order of their cardinality. For example, if there are three joins in a query (e.g. \( J_1, J_2 \) and \( J_3 \)) and the cardinality of \( J_1 \) is 150, \( J_2 \) is 300 and \( J_3 \) is 100 tuples, the join order of greedy solution is \( J_3, J_1 \) and \( J_2 \). In order to select a semi-join program for each join, the cost of having no semi-joins as well as having a beneficial (not gainful) semi-join is calculated. Then, the one which has the lowest cost is selected. In other words, \( C^x \) where \( x \in \{00, 01, 10\} \) is calculated. For example, consider join \( J_1 := R_1.A_1 = R_2.A_1 \); let the cost of having no semi-join be \( C^{00}=200 \)ms, the cost of having \( R_1 \) as the reducer be \( C^{01}=100 \)ms, and the cost of having \( R_2 \) as the reducer be \( C^{11}=250 \)ms; with respect to these costs, the greedy algorithm chooses a semi-join program for \( J_1 \) with \( R_1 \) as the reducer. The join sites and the replica of relations, on the other hand, are chosen randomly since we could not devise a good heuristic for them.

The number of ants is set depending on the number of joins in the query and remains fixed throughout the iterations. For example, if there are five joins in a query, there will be also five JoinAnt, ReplicaAnt, SiteAnt and SemiAnt which construct solutions in each iteration. In other words, there will be four colonies of five ants in each iteration. Table 2 shows the description and the values of MMAS-DJQO parameters which are either suggested in (Dorigo et al. 2004) or found by experiment.

7.2. Solution Construction

Each type of ant constructs a solution based on the random proportional rule of equation below (14). Generally speaking, the probability that ant \( k \), currently in vertex \( i \), chooses to go to vertex \( j \) is (Dorigo et al. 2004)

\[
p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{\ell \in N_i^k} [\tau_{i\ell}]^\alpha [\eta_{i\ell}]^\beta} \quad \text{if } j \in N_i^k ,
\]  

(14)
where $\tau_{ij}$ is the pheromone trail on the edge connecting vertex $i$ to vertex $j$, and $\eta_{ij}$ is the heuristic desirability to choose vertex $j$ when in vertex $i$. Parameters $\alpha$ and $\beta$ respectively control the weight of pheromone trail and heuristic desirability.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>The influence of pheromone</td>
<td>1</td>
</tr>
<tr>
<td>$j\beta$</td>
<td>The influence of heuristic for JoinAnt</td>
<td>2</td>
</tr>
<tr>
<td>$s\beta$</td>
<td>The influence of heuristic for SiteAnt</td>
<td>1</td>
</tr>
<tr>
<td>$\rho$</td>
<td>Pheromone evaporation rate</td>
<td>0.02</td>
</tr>
<tr>
<td>ANTS-N0</td>
<td>The number of ants</td>
<td>The number of joins</td>
</tr>
<tr>
<td>MAX-ITERATION</td>
<td>The maximum number of iterations</td>
<td>20</td>
</tr>
<tr>
<td>RESET-COUNT</td>
<td>The number of consecutive iterations without improvement</td>
<td>10</td>
</tr>
</tbody>
</table>

The heuristic desirability and the value of $\beta$ are different for each type of ant. For the JoinAnt, the heuristic desirability is $\eta_{ij} = \frac{1}{\text{card}_{j}}$, which means that the desirability to select $j$th join ($J_j$) is the inverse of the cardinality of the intermediate relation produced by $J_j$. In other words, the smaller the cardinality of intermediate relation produced by a join is, the more desirable it is. The value of $j\beta$ (i.e. $\beta$ for JoinAnt) is set to 2. SiteAnt uses this heuristic: if the cardinality of the intermediate relation is smaller than sum of the cardinalities of involved relations, the site which holds the bigger relation’s replica has more chance than the other sites (in this way, the bigger relation need not to be transmitted so the communication costs are reduced). Otherwise, all the sites have the same chance of being selected. The value of $j\beta$ (i.e. $\beta$ for SiteAnt) is set to 1. ReplicaAnt and SemiAnt use no heuristics. Hence, the replicas and the semi-joins are selected only based on pheromone trails.

It should be noted that at an early stage of our experiments, we used a heuristic for SemiAnt which was the inverse of the $C^x$, $x \in \{00, 01, 10\}$. In other words, the cost of having no semi-join ($x = '00'$) or having either of the relations as the reducer ($x = '01'$ or $x = '10'$) was calculated. The lower the cost was, the more desirable was for the ant to select it. However, this heuristic caused the algorithm to get stuck in local optimum even with low influence of $\beta$. 

22
Notice that SiteAnt should start its task after JoinAnt and ReplicaAnt because the
heuristic of SiteAnt uses the order of the joins and the sites at which replicas are
held, produced by JoinAnt and ReplicaAnt, respectively.
In equation (14), the feasible set of neighbors of ant $k$ currently in vertex $i$, is
represented by $N_i^k$. For the JoinAnt, this set is composed of all unvisited joins;
for the SiteAnt, it is made up of all sites because joins can be executed at every
site; for the ReplicaAnt, the set includes all sites holding a replica of a relation
(correspond to vertex $i$) and finally for SemiAnt, the set is \{00, 01, 10\}.
The sum of JoinAnt, ReplicaAnt, SiteAnt and SemiAnt solutions produce a QEP.
When all ants construct their QEPs, their costs are evaluated based on the cost
model. The cost of the iteration-best solution is also compared to the best-so-far
solution and if the cost proves to be lower, will be replaced by it.
One of the characteristics of MMAS algorithm is that pheromone values are
limited to the range $[\tau_{min}, \tau_{max}]$. Where the upper pheromone trail limit is set to
$$\tau_{max} = \frac{1}{\rho \cdot C^{bs}}$$
the $C^{bs}$ is the cost of best-so-far solution. The lower pheromone
trail limit, $\tau_{min}$, is related to $\tau_{max}$ (i.e. $\tau_{min} = a \cdot \tau_{max}$). In MMAS-DJQO, $a$ is
set to 0.5. Another characteristic of MMAS algorithm is re-initialization of
pheromone trails to prevent stagnation. In MMAS-DJQO algorithm, if the best-so-
far solution is not changed for a certain number of consecutive iterations
(RESET-COUNT = 10), the pheromone is re-initialized to $\tau_{max}$.

7.3. Pheromone Update

When the ants construct their solutions, pheromone update is performed. First, pheromone trails on all edges of all graphs are evaporated based on the following
equation (Dorigo et al. 2004):
$$\tau_{ij} \leftarrow (1 - \rho) \tau_{ij} \quad \forall (i, j) \in A$$
(15)
where $\rho$ is pheromone evaporation rate and $A$ is the set of all vertices of problem
graphs. Then, the JoinAnt, ReplicaAnt, SiteAnt and SemiAnt, which have found the best-so-far solution, deposit pheromone on the edges they visited
(τ_{ij} ← τ_{ij} + \frac{1}{C_{hs}}). This deposited pheromone directs the ants’ search in the next iteration.

### 7.4. Local Search

After all iterations are passed and before outputting the best solution, a part of its neighborhood is investigated with the hope to find a better solution. The neighborhood is produced by changing the join site and semi-join of each join. The cost of choosing other sites and other semi-joins is evaluated for every join, and then the one with the least cost is selected. Since we did not want to increase optimization time, the join permutation and the selected replicas of relations remain intact.

According to our experiments, this local search step in MMAS-DJQO improves the quality of solutions by 15.24%; though, it causes the running time to increase by 6.55%. Hence, the overall efficiency of the algorithm improves. If we had added this local search at each of the iterations, the optimization time would have been significantly increased; therefore, it would not be acceptable for a query optimization algorithm.

### 8. Alternate algorithms for comparison

The proposed algorithm is compared with two previous genetic-based algorithms—OGA97, which could be executed in replicated databases, and NGA11, which can be used in non-replicated databases. As mentioned in section 2, we do not intend to compare our algorithm to those that only restrict the search space to find the best execution order. In this section, a short description of these two algorithms is presented.

#### 8.1. OGA97 Algorithm

In OGA97, each chromosome consists of four genes. Each gene corresponds to a decision for selecting the optimum plan. The chromosome structure is defined as follows:

<table>
<thead>
<tr>
<th>Copy Id</th>
<th>Join Site</th>
<th>Semi-join</th>
<th>Join Order</th>
</tr>
</thead>
</table>

24
The *Copy Id* gene is an array with a position for each relation. Each value in the array represents the site at which the chosen replica of the relation is stored. *Join site* gene is an array with a position for each join. Each value in this array is the site at which the corresponding join is executed. *Semi-join* gene is a set of 2 bits. Each set corresponds to a join in the query. The value of the bit corresponding to the reducer file is either set to 1 or it is 0 (just elementary semi-joins are considered). *Join order* gene is a permutation of joins which shows the order in which they are performed. The pseudo-code of OGA97 is shown below:

```
(1) pop-size = 200; Iteration-No = 2000; mutation-rate = 0.005
(2) Generate feasible QEPs randomly
(3) Calculate the fitness of QEPs.
(4) For iter = 1 to Iteration-No
(5) Select two parents based on roulette wheel selection method
(6) Create a new child by applying crossover and mutation
(7) If the child is fitter (has lower cost) than the worst in population
(8) Replace the worst with the new child.
(9) End If
(10) End for
(11) Output the fittest in population.
```

In this algorithm the population size (pop-size) is set at 200, and the mutation rate (mutation-rate) at 0.005. A single new child is created in each iteration to replace the worst member of the old population. The algorithm stops when the number of iterations (Iteration-No) reaches 2000 (Rho et al. 1997).

### 8.1.1. Crossover

OGA97 uses uniform crossover for *copy Id*, *semi-join* and *join site* genes. In uniform crossover, values are copied to child from the first or second parent with equal probability. For *join order* gene, uniform order crossover is applied which produces valid solutions for the permutation representation.

### 8.1.2. Mutation

For *Copy Id*, *Semi-join* and *join site* genes, a typical mutation is applied, which randomly changes one of the values in these genes. For the *join order* gene, inversion operator is alternatively used, which randomly selects two points and reverses the order of values between them.
8.2. NGA11 Algorithm

This algorithm is recently proposed for DJOQ. Each chromosome consists of \( n \) genes with three parts \( (n \) is the number of joins in the query): Condition number, Node number and semi-join. Node number refers to the site at which the join, which is represented by condition number, is performed. A random chromosome structure for a query with 3 joins is shown below:

![Diagram](image)

The order of genes in chromosome shows the join order. Each gene is associated with the cost of executing the corresponding join at the site showed by node number with respect to the type of semi-join. The pseudo-code of NGA11 is shown below:

1. pop-size = 100; Xover-rate = 0.6; mutation-rate = 0.015;
2. Generate initial population randomly;
3. Do
4. Evaluate the cost of each gene and chromosome;
5. Produce new population by applying new-crossover and new-mutation
6. Until 95% of the chromosomes are not changed in a new generation;
7. Output the best chromosome;

In this algorithm, the population size \( \text{pop-size} \) is set at 100, while the rates of the crossover \( \text{Xover-rate} \) and the mutation \( \text{mutation-rate} \) are set at 0.6 and 0.015 respectively. The algorithm terminates when 95% of chromosomes are not altered in the new generation (Sevinç et al. 2011).

8.2.1. Crossover

This algorithm introduces a new crossover which finds the \( k \)-length least cost subsequence of genes from one parent first, and put them in the child with the same order. The rest of the genes are copied from the other parent while preserving the order of their appearance. \( k \) is the number of genes in the chromosome multiplied by the crossover rate. Hence, if there are 8 genes in the chromosome and the crossover rate is 0.6, the new crossover operator looks for a 5-length subsequence of genes with the minimum cost.
The focus of the new crossover in NGA11 is the join order. It does not change the node number or the semi-join. However, a gene, which corresponds to a join, might have a high cost due to the assigned node number or semi-join.

8.2.2. Mutation

The new mutation introduced in NGA11, selects a gene with a probability proportional to its cost, and changes its node number and semi-join. Hence, the gene with the highest cost (the worst gene) has the highest chance of being selected. The mutation rate is 0.015.

8.3. Modified NGA11 algorithm (MO-NGA)

As the chromosome structure of NGA11 shows, this algorithm does not consider any decision on replicated relations (i.e. it assumes either data is stored non-redundantly or relation replicas have been pre-selected). We modify this algorithm so as for it to be applicable in replicated databases. For this purpose, another part has to be added to each gene which corresponds to chosen replicas of relations involved in the join. The added part (i.e. selected replicas) represents the sites where the relations involved in the join are held. Thus, the chromosome structure is changed as follows:

<table>
<thead>
<tr>
<th>Condition Number</th>
<th>Node Number</th>
<th>Semi-join</th>
<th>Selected replicas</th>
</tr>
</thead>
</table>

Through this representation, a different replica of a relation may be chosen for different joins. For example, if $R_2$ is involved in both $J_1$ and $J_2$, a different replica of $R_2$ may be selected for each of the joins in accordance with this structure. As in OGA97 and the proposed algorithm, only one replica of a relation is used in all joins; we restrict the random generation of chromosomes to allow just one replica of a relation in all joins. This modified version of NGA11, which we call MO-NGA, uses the same crossover; but the mutation operator changes selected replicas as well as other parts of a gene.

9. Experimental Evaluation

In order to study the efficiency of our proposed algorithm, several experiments are carried out which are explained in this section. As stated earlier, the MMAS-
DJQO algorithm is compared with NGA11 and OGA97 algorithms in both replicated and non-replicated environments. MO-NGA algorithm is used in the replicated environments, while in non-replicated environments, copy Id gene is removed from OGA97 and ReplicaAnt is removed from our proposed algorithm. In a comparison made between OGA97 algorithm and exhaustive search algorithm on chain, tree and cyclic queries, the reported results showed the possibility of finding good plans in a reasonable amount of computing time (Rho et al. 1997). This old algorithm is the most complete algorithm for replicated databases which could be found in the literature in order to perform the comparison. NGA11 algorithm, on the other hand, has been consecutively compared with a random algorithm, exhaustive search as well as another genetic algorithm on a synthetic database with six relations and has ended up with better results. It has been only experimented on linear queries though (Sevinç et al. 2011).

We have conducted four experiments, the first two are similar to those performed on OGA97 algorithm and the latter ones are similar to those suggested in NGA11 algorithm. The purpose is to study the trade-off between optimization time and the optimality of the solutions (QEPs). In other words, our goal is to minimize the query evaluation time, which is the sum of QEP’s executing time and optimization time (Yu et al. 1997).

### 9.1. The hardware/software configuration

The experiments are all done in an environment using a 2 GHz AMD Athlon Dual-core processor with 3 Giga bytes of DDR2 RAM with Microsoft Windows Vista Home Premium as the operating system. All the algorithms (i.e. OGA97, NGA11, MO-NGA and MMAS-DJQO algorithms) were implemented in the programming environment provided by MATLAB R2008a. The algorithms are tested by the same queries generated randomly by our problem generator which is explained in the next sub section.

### 9.2. Problem Generator

Our problem generator consists of three parts:
(1) The database generator takes the number of relations as input in order to generate a synthetic database. It determines the cardinality of relations, the average length of tuples in each relation, the join attribute length and the number of distinct values in each join attribute. The relation cardinalities, average tuple sizes and the sizes of join attributes are randomly selected in range [300,700], [15, 90] and [2, 8] respectively. For example, if the number of relations is set at four, it first generates a schema with four relations \( R_1(A_1, A_5, A_6), R_2(A_1, A_2, A_6), R_3(A_3, A_6) \) and \( R_4(A_3, A_5, A_6) \), for each of which, it randomly generates metadata information afterward. For instance, it may set \( R_1 \)'s cardinality to 450 tuples, its average tuple size to 40 bytes and the size of join attributes to 6 bytes.

(2) The replica generator takes replication degree as well as the number of sites as inputs and replicates each relation according to that degree. For example, if the degree is four, each relation is randomly replicated in at most four sites. In other words, the degree of replication determines the maximum number of sites at which a relation could be replicated. Regarding non-replicated databases, the degree is set to one so that each relation is held at only one site.

(3) The query generator takes the number of joins as input and generates a chain query based on the schema. For example, it may generate a query with three joins (e.g. \( J_1, J_2 \) and \( J_3 \)). It also determines the purity of the join attributes and the result site.

In a nutshell, our problem generator produces queries on a synthetic database. The random nature of our problem generator uses more generalized queries than those of a specific application. Thus, the reported results are not dependent on any specific database.

The page size is set to 10240 bytes, the disk I/O time is 10ms (per page) at all sites and the transfer time is 1Gbps.

### 9.3. Experimental Results

In this section, we explain four different experiments which are carried out to compare the implemented algorithms. The descriptions of these experiments are shown in Table 3. In all the experiments, each query is solved 30 times and the average QEP quality and optimization time are calculated. The results of the first two experiments have been illustrated in tabular form similar to OGA97, while
those of the last two experiments have been better to be depicted in a graphical form due to their nature.

**Table 3** Descriptions of experiments

<table>
<thead>
<tr>
<th>Experiment No.</th>
<th>Goal</th>
<th>Type of join query</th>
<th>Environment</th>
<th>Cost model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experiment 1</td>
<td>To compare the optimal QEP with the QEP found by the implemented algorithms</td>
<td>Chain queries</td>
<td>Without replication</td>
<td>Total time Response time</td>
</tr>
<tr>
<td>Experiment 2</td>
<td>To compare OGA97 algorithm with the proposed algorithm on tree and cyclic queries</td>
<td>Tree and cyclic queries</td>
<td>Replicated</td>
<td>Total time</td>
</tr>
<tr>
<td>Experiment 3</td>
<td>To compare the implemented algorithms by fixing the No. of sites and varying the No. of joins</td>
<td>Chain queries</td>
<td>Replicated</td>
<td>Total time Response time</td>
</tr>
<tr>
<td>Experiment 4</td>
<td>To compare the algorithms by fixing the No. of joins and varying the No. of sites.</td>
<td>Chain queries</td>
<td>Replicated</td>
<td>Total time Response time</td>
</tr>
</tbody>
</table>

**Experiment 1:**

In this experiment, first, simple queries (i.e. queries with a few numbers of joins) are generated in a non-replicated environment so as to be solved by an exhaustive search algorithm. Table 4 shows the optimum solution of these queries when the cost model is either based on response time or total time. In this table, *chain*  

\[ (xJ,yS) \]

means a chain query with *x* number of joins which is executed in a distributed database with *y* number of sites. Then, these queries are solved by the implemented algorithms. The result of this experiment is shown in Table 5. The results show that OGA97 algorithm has the least relative error\(^4\) compared to other algorithms (the average relative error of this algorithm is 0.64%). The average relative errors of MMAS-DJQO and NGA11 are 4.57% and 34.39% respectively. However, the optimization time of OGA97 is higher when compared to our proposed algorithm. In fact, our proposed algorithm saves up 90% of optimization time compared to OGA97 algorithm in this experiment.

\(^4\) The relative error is the absolute error divided by the magnitude of the exact value.
A good optimizer should make a reasonable trade-off between generated QEP quality and optimization time so that executing a sub-optimal QEP does not exceed the optimization time. Hence, when the purpose is to minimize query evaluation time, which is the sum of the time to execute the query and the optimization time, the proposed algorithm is faster than others.

Notice that in Table 5, NGA11 is removed from the comparison when the cost model is based on the response time. The reason is that NGA11 uses problem-specific crossover and mutation and was originally developed with a total time cost model. On the other hand, although the original version of OGA97 uses a cost model based on total time, due to the application of a general crossover and mutation, the algorithm is not dependent on its cost model. Hence, the cost model could be replaced easily.

<table>
<thead>
<tr>
<th>Query Type</th>
<th>Optimum QEP cost (Total Time)</th>
<th>Optimum QEP cost (Response Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chain(3J,2S)</td>
<td>0.9122</td>
<td>0.5920</td>
</tr>
<tr>
<td>Chain(4J,3S)</td>
<td>0.5608</td>
<td>0.2907</td>
</tr>
<tr>
<td>Chain(5J,3S)</td>
<td>0.6402</td>
<td>0.3100</td>
</tr>
<tr>
<td>Chain(6J,2S)</td>
<td>3.4825</td>
<td>2.0800</td>
</tr>
</tbody>
</table>

**Experiment 2**

Our query generator generates only chain queries; however, since OGA97 algorithm had been earlier experimented on tree and cyclic queries, in this experiment, we try to compare our proposed algorithm with OGA97 by applying it to the query graphs it had been tested on (Rho et al. 1997), which are shown in Fig. 4.

The results of experiment 2 are shown in Table 6. Accordingly, although the proposed algorithm cannot find any better solution quality than OGA97 algorithm, it saves up more than 88% of optimization time and has the least query evaluation time.
Table 5 The results of experiment 1

<table>
<thead>
<tr>
<th>Algorithm name</th>
<th>Experimental results</th>
<th>Chain(3J,2S)</th>
<th>Chain(4J,3S)</th>
<th>Chain(5J,3S)</th>
<th>Chain(6J,2S)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OGA97</strong></td>
<td>Total Time</td>
<td>Avg. QEP Cost</td>
<td>0.9122</td>
<td>0.5618</td>
<td>0.6524</td>
</tr>
<tr>
<td></td>
<td>Avg. Opt. Time</td>
<td>0.5068</td>
<td>0.5405</td>
<td>0.5716</td>
<td>0.6109</td>
</tr>
<tr>
<td></td>
<td>Relative Error</td>
<td>0%</td>
<td>0.18%</td>
<td>1.91%</td>
<td>0.49%</td>
</tr>
<tr>
<td><strong>Response Time</strong></td>
<td>Avg. QEP Cost</td>
<td>0.5920</td>
<td>0.2935</td>
<td>0.3172</td>
<td>2.1082</td>
</tr>
<tr>
<td></td>
<td>Avg. Opt. Time</td>
<td>0.6691</td>
<td>0.7170</td>
<td>0.7878</td>
<td>0.8240</td>
</tr>
<tr>
<td></td>
<td>Relative Error</td>
<td>0%</td>
<td>0.96%</td>
<td>2.32%</td>
<td>1.36%</td>
</tr>
<tr>
<td><strong>NGA11</strong></td>
<td>Total Time</td>
<td>Avg. QEP Cost</td>
<td>1.1309</td>
<td>0.8039</td>
<td>0.9578</td>
</tr>
<tr>
<td></td>
<td>Avg. Opt. Time</td>
<td>0.3186</td>
<td>0.4272</td>
<td>0.5135</td>
<td>0.5174</td>
</tr>
<tr>
<td></td>
<td>Relative Error</td>
<td>23.98%</td>
<td>43.35%</td>
<td>49.61%</td>
<td>20.64%</td>
</tr>
<tr>
<td><strong>MMAS-DJQO</strong></td>
<td>Total Time</td>
<td>Avg. QEP Cost</td>
<td>0.9236</td>
<td>0.5768</td>
<td>0.7076</td>
</tr>
<tr>
<td></td>
<td>Avg. Opt. Time</td>
<td>0.0216</td>
<td>0.0329</td>
<td>0.0467</td>
<td>0.0669</td>
</tr>
<tr>
<td></td>
<td>Relative Error</td>
<td>1.25%</td>
<td>2.85%</td>
<td>10.53%</td>
<td>3.64%</td>
</tr>
<tr>
<td><strong>Response Time</strong></td>
<td>Avg. QEP Cost</td>
<td>0.6040</td>
<td>0.3084</td>
<td>0.3284</td>
<td>2.2211</td>
</tr>
<tr>
<td></td>
<td>Avg. Opt. Time</td>
<td>0.0312</td>
<td>0.0447</td>
<td>0.0599</td>
<td>0.0853</td>
</tr>
<tr>
<td></td>
<td>Relative Error</td>
<td>2.03%</td>
<td>6.09%</td>
<td>5.94%</td>
<td>6.78%</td>
</tr>
</tbody>
</table>

Table 6 The results of experiment 2

<table>
<thead>
<tr>
<th>Algorithm name</th>
<th>Experimental results</th>
<th>Tree(4J,4S)</th>
<th>Tree(10J,8S)</th>
<th>Cyclic(4J,5S)</th>
<th>Cyclic(6J,5S)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OGA97</strong></td>
<td>Total Time</td>
<td>Avg. QEP Cost</td>
<td>0.7063</td>
<td>3.4542</td>
<td>0.7008</td>
</tr>
<tr>
<td></td>
<td>Avg. Opt Time</td>
<td>0.5751</td>
<td>0.8235</td>
<td>0.5827</td>
<td>0.6561</td>
</tr>
<tr>
<td></td>
<td>Evaluation Time</td>
<td>1.2814</td>
<td>4.2777</td>
<td>1.2835</td>
<td>1.1397</td>
</tr>
<tr>
<td><strong>MMAS-DJQO</strong></td>
<td>Total Time</td>
<td>Avg. QEP Cost</td>
<td>0.7410</td>
<td>3.8910</td>
<td>0.7998</td>
</tr>
<tr>
<td></td>
<td>Avg. Opt Time</td>
<td>0.0386</td>
<td>0.1090</td>
<td>0.0388</td>
<td>0.0755</td>
</tr>
<tr>
<td></td>
<td>Evaluation Time</td>
<td>0.7796</td>
<td>4.0000</td>
<td>0.8386</td>
<td>0.5591</td>
</tr>
</tbody>
</table>
Fig 4. Query graphs of experiment 2 (which OGA97 algorithm had been tested on)

**Experiment 3**

In this experiment, the number of sites is fixed by 4 and the number of joins varies from 3 to 15\(^5\). The queries are generated in a replicated environment with the degree of four. Thus, each relation could be replicated at maximum four sites. The modified version of NGA11 (i.e. MO-NGA) is used to support replication.

This experiment is divided into two parts: in the first part (experiment 3(a)), the cost model is based on the total time and in the second part (experiment 3(b)), it is based on the response time.

- **Experiment 3(a):** In this experiment, a comparison has been made between OGA97, MO-NGA and MMAS-DJQO algorithms, where the cost model is based on the total time. Fig. 5 shows the optimality of generated QEPs (i.e. QEP quality). Fig. 6 demonstrates the optimization

---

\(^5\) There are database system applications that contain complex queries with a large number of joins (sometimes even up to 100). There are also business data processing applications (such as banking, ticket reservation and inventory management) that involve queries with smaller number of joins (usually less than 10). The maximum number of joins in OGA97 and NGA11 experiments was 10 and 6 respectively. We use the maximum of 15 join conditions in our experiment which seems to be quite sufficient.
time. As this figure shows, the optimization time increases along with the growing number of joins. Fig. 7 presents the query evaluation time (i.e. the sum of query execution time and optimization time). According to these figures, it can be concluded that:

(1) Although MO-NGA generates the worst QEPs, its optimization time is less than OGA97 algorithm when the number of joins is less than 6.

(2) MMAS-DJQO algorithm produces a little worse QEPs (with the difference of 11.55%), but it saves optimization time by 78.79% compared to OGA97.

(3) MMAS-DJQO improves the average of query evaluation time of OGA97 and NGA11 by 26.13% and 47.31%, respectively.

Fig. 5 Average QEP quality in experiment 3(a)
- **Experiment 3(b):** This experiment is conducted on the same queries as experiment 3(a), though, with the cost model based on the response time. MO-NGA algorithm is removed from this experiment because its problem-specific crossover and mutation cannot be used with a response time cost model. The average optimality of the QEPs, optimization time and query evaluation time are
illustrated in figures 8, 9, 10 respectively. From these figures, we can conclude that:

(1) OGA97 algorithm produces better QEPs than MMAS-DJQO with the difference of 11.09%. However, MMAS-DJQO saves optimization time by 81.91%.

(2) MMAS-DJQO algorithm improves the average evaluation time of OGA97 algorithm by 52.12%.

**Experiment 4**

In this experiment, a chain query with six joins is generated in a replicated environment. The number of sites in this environment, which is considered as the degree of replication, is varied from 1 (i.e. a centralized database system) to 8. For example, when there are five sites, the degree of replication is also five and when there are eight sites, the degree is eight. According to the cost model and similar to the previous experiment, this experiment is also divided into two parts—experiment 4(a) and experiment 4(b).

![Fig. 8 Average QEP quality in experiment 3(b)](image-url)
Fig. 9 Average optimization time in experiment 3(b)

Fig. 10 Average query evaluation time in experiment 3(b)

- **Experiment 4(a):** The cost model is based on the total time. Fig. 11 illustrates the optimality of QEPs. Note that because of the absence of any communication costs, the least total time is achieved when there is only one site. Once the number of sites grows, due to having more parallel
local processing and transmissions, the total time will also increase. Fig. 12 shows the optimization time which does not change considerably when the number of sites increases. Finally, Fig. 13 depicts the query evaluation cost. Based on these figures, it can be concluded that:

1. MO-NGA algorithm produces the worst QEPs, but its optimization time is less than OGA97 in this experiment.
2. MMAS-DIJO algorithm generates worse QEPs than OGA97 with the difference of 8.38%, but saves up 88.41% of optimization time.
3. MMAS-DIJOQ improves the average of query evaluation time of OGA97 and NGA11 by 36.47% and 50.75%, respectively.

![Graph](image)

**Fig. 11** Average QEP quality in experiment 4(a)
Fig. 12 Average optimization time in experiment 4(a)

Fig. 13 Average query evaluation time in experiment 4(a)

- **Experiment 4(b):** this experiment is done on the same query as experiment 4(a), though with the response time cost model. The optimality of QEPs is illustrated in Fig. 14. The shift in the number of sites from 1 to 2 brings about the feasibility of parallel execution which leads to a considerable reduction in query execution time. The optimization overhead is shown in Fig. 15 and the query evaluation time is depicted in Fig. 16. From these figures, we can conclude that:
(1) OGA97 algorithm produces better QEPs than MMAS-DJQO with the difference of 25.39%. However, MMAS-DJQO saves optimization time by 89.52%.

(2) MMAS-DJQO algorithm improves the average evaluation time of OGA97 algorithm by 51.07%.

Fig. 14 Average QEP quality in experiment 4(b)

Fig. 15 Average optimization time in experiment 4(b)
According to experiments 3 and 4, OGA97 algorithm outperforms MO-NGA and MMAS-DJQO algorithms in terms of QEP quality. Nevertheless, MMAS-DJQO algorithm has the least optimization overhead. The preference between these algorithms depends on the intended application area. When the queries are ad-hoc (i.e. non-repetitive), due to its least optimization overhead, our MMAS-DJQO algorithm can be a more appropriate choice than OGA97 and NGA11 algorithms. On the other hand, if the queries are executed frequently (i.e. repetitive query), since the optimization cost can be amortized over multiple executions, the quality of generated solutions will be a more important characteristic and OGA97 will make a better choice.

10. Conclusion and Future work

Query optimization is the most important phase in query processing. In distributed database systems, the submitted query is decomposed to a number of sub-queries which execute at different sites. Hence, in addition to local optimization which takes place at each site-with the same algorithms used in centralized database system- there is a global optimization step which selects an optimum strategy to execute the query. In order to find the optimum strategy, we use the comprehensive model introduced in OGA97 algorithm, in which four decisions are integrated: (1) replica selection, (2) join site selection, (3) join order and (4) reduction by semi-join. Due to combinatorial search space, finding the optimum
strategy is computationally intractable. Hence, the real goal of the optimizer is to find a near optimum strategy and -more importantly- to avoid bad strategies.

In this paper, we propose an ACO algorithm for distributed join query optimization which is based on MAX-MIN Ant System. As in real ant colonies each individual ant is not responsible for all the existing tasks, in our proposed algorithm, four types of ants are used for each of the decisions made in the model. This increases modularity and scalability. For example, in non-replicated environments, ReplicaAnt- which is in charge to select an appropriate replica of a relation in order to be used in join operations-could be removed. In addition, new heuristics could be devised for each of the ants separately.

The proposed algorithm is then compared to two previous genetic-based algorithms with a total time as well as response time cost model on chain, cyclic and tree queries. The experimental results show that our MMAS-DJQO algorithm improves the average optimization time by nearly 80%; however, it produces worse QEPs compared to OGA97 algorithm with an average of 14.31% reduction in the optimality of solutions. To conclude, our algorithm makes a good compromise between the time to determine a near-optimum plan and the quality of the plan. Therefore, compared to OGA97 and NGA11 algorithms, the cost of evaluating a query is minimized. MMAS-DJQO, among these three algorithms, is more appropriate for ad-hoc queries- which are not repetitive- due to its little optimization overhead. For frequently executed queries, on the other hand, QEP quality is more important and OGA97 is a better choice.

There are several major areas for future studies. First, by using learning automata (Narendra et al. 1974), the proposed algorithm could be improved to adjust the algorithm parameters. This way seems to be more promising than finding the parameters by trial and error. Second, other newer swarm intelligence algorithms like Intelligent Water Drops (IWD) (Shah-Hosseini 2009) could also be applied to the DJQO problem. Moreover, a hybrid algorithm of IWD and ACO, which use the strengths of both algorithms, could be applied as well. Third, in this paper only join queries are considered. Similar algorithms could be developed for other types of queries such as set queries which play an important role in horizontally fragmented databases. Finally, since in our proposed algorithm fragmentation is not exploited, future research should consider the development of a heuristic algorithm for a fragmented database.
References


DATE C. J., (2004), An Introduction to Database Systems, 8th ed, Addison-Wesley, USA.


*Distributed and Parallel Databases, 5*: 5-30.

Author Biographies

**Ladan Golshganara** received her Bachelor's and Master's degree in Computer Engineering-Software from Shahid Bahonar University and Shahid Beheshti University in 2009 and 2011 respectively. Her research interests include Query processing and Optimization, Distributed and Parallel Databases, Swarm intelligence and Ontologies.

**Seyed Mohammad Taghi Rouhani Rankoohi** received his bachelor's degree in mathematics from Tehran University, Bachelor's and Master's degree in informatics, and the D.E.S.S degree in Teleinformatics from Pierre and Marie Curie University in Paris. He is an associate professor at Shahid Beheshti University (SBU). He has mostly lectures on File Engineering and Databases at SBU and other universities such as Sharif University of Technology and Tehran University. He received "outstanding lecturer" award in SBU in 2007. His publications include several papers published in Iranian journals (such as Scientia Iranica) and conference proceedings, four translations of textbooks into Farsi, as well as eight authored books, some of which are widely used as textbooks in Iran. He has received the "Book of the Year" award in 2003 and the "appreciable book" award in the book-of-the-year ceremony in 1994, 2009, 2010 and 2012. Two of his books have also been honored as the "Selected Academic book of the Year" by Tehran University in 1992 and 2002. He is co-author of two papers published in KAIS in the field of distributed database systems. His research interests include Database Systems, Multi-database integration and File Engineering. Reading modern literature is among his hobbies.
Hamed Shah-Hosseini was born in Tehran, Iran, in 1970. He received the B.S. degree in Computer Engineering from the University of Tehran. He also obtained the M.S. and Ph.D. degrees both in Computer Science (AI) from the Amirkabir University of Technology, all with high honors.

His research interests include Computational Intelligence especially Neural Networks, Evolutionary Computation, Swarm Intelligence, and Computer Vision. He proposed the Time-Adaptive Self-Organizing Map (TASOM) networks for both stationary and non-stationary environments. Moreover, he introduced two bio-inspired swarm-based optimization algorithms: “Intelligent Water Drops” algorithm (or IWD algorithm) and “Galaxy-based Search Algorithm” (or the GbSA). In addition, he presented a Generalized Taylor’s (G-Taylor’s) Theorem.