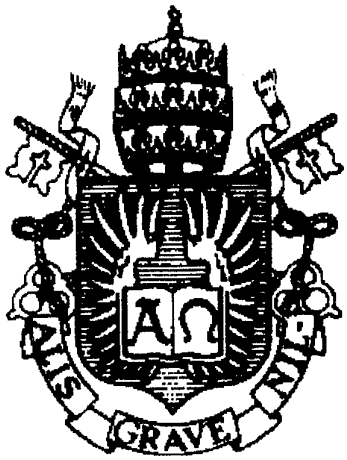
PUC



ISSN 0103-9741

Monografias em Ciência da Computação
nº 05/95

Query Optimization in Distributed Relational Databases

Rosana S. G. Lanzelotte
Celso C. Ribeiro
Cláudio D. Ribeiro

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, Nº 05/95

Editor: Carlos J. P. Lucena March, 1995

Query Optimization in Distributed Relational Databases*

Rosana S. G. Lanzelotte

Celso C. Ribeiro

Cláudio D. Ribeiro

* This work has been sponsored by the Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio — Departamento de Informática
Rua Marquês de São Vicente, 225 — Gávea
22453-900 — Rio de Janeiro, RJ
Brasil
Tel. +55-21-529 9386 Telex +55-21-31048 Fax +55-21-511 5645
E-mail: rosane@inf.puc-rio.br

Query Optimization in Distributed Relational Databases

Rosana S.G. Lanzelotte (rosana@inf.puc-rio.br)
Celso C. Ribeiro (celso@inf.puc-rio.br)
Cláudio D. Ribeiro (degrazia@inf.puc-rio.br)

Pontifícia Universidade Católica do Rio de Janeiro
Departamento de Informática
Rua Marquês de São Vicente 225
Rio de Janeiro 22453
Brazil

PUCRioInf-MCC05/95

January 1995

Abstract

The query optimizer is the DBMS (data base management system) component whose task is to find an optimal execution plan for a given input query. Typically, optimization is performed using dynamic programming. However, in distributed execution environments, this approach becomes intractable, due to the increase in the search space incurred by distribution. We propose the use of the tabu search metaheuristic for distributed query optimization. A hashing-based data structure is used to keep track of the search memory, simplifying significantly the implementation of tabu search. To validate this proposal, we implemented the tabu search strategy in the scope of an existing optimizer, which runs several search strategies. We focus our attention into the more difficult problems in terms of the query execution space, in which bushy execution plans and Cartesian products are accepted, which are not dealt with very often in the literature. Using a real-life application, we show the effectiveness of tabu search when compared to other strategies.

Keywords: Query optimization, relational databases, distributed systems, tabu search

Resumo

O otimizador de consultas é o componente de um SGBD (sistema de gerência de banco de dados) responsável por obter um plano de execução ótimo para uma dada consulta. Programação dinâmica é a ferramenta tipicamente utilizada no processo de otimização. Entretanto, em ambientes de execução distribuídos este enfoque torna-se intratável, devido ao aumento do espaço de busca devido à distribuição. Neste artigo propõe-se a utilização da metaheurística busca tabu para a otimização de consultas distribuídas. A memória da busca é implementada através de uma estrutura de dados baseada em *hashing*, simplificando significativamente a implementação da busca tabu. Para validar esta proposta, a estratégia de busca tabu foi implementada dentro de um otimizador já existente, que trata diferentes estratégias de busca. Este estudo concentra-se em problemas mais difíceis em termos do espaço de execução da consulta, nos quais os planos de execução podem ser representados por árvores binárias quaisquer e produtos cartesianos são aceitos, que vêm sendo pouco tratados na literatura. Com base em uma aplicação real, ilustra-se a eficiência da busca tabu, quando comparada com outras estratégias.

Palavras-chaves: Otimização de consultas, bancos de dados, sistemas distribuídos, busca tabu

1 Introduction

Query optimization refers to the process of building an optimal execution plan for a given query, with respect to a cost function to be minimized. This is a difficult task due to the trade-off between the optimization cost and the quality of the generated plans (the latter translates into the response time, or query execution cost). The cost of optimizing a query is mainly incurred by the investigation of the solution space for alternative execution plans. The size of the solution space depends on the complexity of the input query and also on the alternatives offered by the execution environment. In a centralized environment, an execution plan is run by a single server, while in a distributed one it is possible to exploit concurrent execution. Thus, many more execution alternatives exist in the latter case, increasing the solution space when compared to a centralized environment. A high optimization cost may be acceptable for a repetitive query, since it can be amortized over multiple executions. However, it is not practical for *ad-hoc* queries that are executed only once.

As the solution space gets larger for complex queries or distributed environments, the search strategy that investigates alternative solutions is critical for the optimization cost. Although dynamic programming is traditionally used for finding the exact optimal solution to query optimization problems, this strategy faces a combinatorial explosion for complex queries (typically, a join query with more than ten relations). In order to investigate larger spaces, local search strategies have been proposed in the litterature to improve an initial solution until obtaining a local optimum. Examples of such strategies are simulated annealing [17] and iterative improvement (hill-descending) [24]. These strategies do not guarantee that the best solution is obtained, but they are able to reduce the optimization cost.

In this paper, we propose the tabu search metaheuristic as an alternative approach for finding good approximate solutions for query optimization. Tabu search is an adaptive procedure for solving combinatorial optimization problems, which guides a hill-descending heuristic to continue exploration without becoming confounded by the absence of improving moves, and without falling back into a local optimum from which it previously emerged. Although more time consuming than other local search schemes such as hill-descending and simulated annealling, many applications of tabu search reported in the litterature have shown that the latter consistently leads to better solutions.

Previous applications of the tabu search metaheuristic implemented a short-term search memory either by keeping a circular list with the most recently visited solutions, or by recording a list of *tabu-active* solution characteristics which may not appear in any new solution during a certain number of iterations. Another contribution of this paper concerns the use of a hashing-based data structure that enables storing the entire history of the search, i.e., all solutions visited by the algorithm, considerably simplifying the algorithm implementation.

We validate our proposal by implementing the tabu search metaheuristic in the scope of an existing query optimizer, originally built for the EDS ESPRIT project [4], which runs several search strategies [20]. We compare the quality of the solutions obtained by tabu search with the optimal solutions obtained by dynamic programming for small- and medium-size problems. Comparisons for large-size problems, when dynamic programming is no more feasible in computational terms, are made with the solutions obtained by other local search algorithms.

The computational experiments were carried out using a real-life testbed, corresponding to an application of PETROBRAS, the Brazilian state-owned petroleum company in charge of oil

prospection, exploitation and distribution. The application refers to geological research for locating new wells and finding oil, dealing with seismical data, oil basins and technician teams in charge of these investigations.

The paper is organized as follows. In Section 2, we give the detailed formulation of the query optimization problem, as well as the description of the execution space and the cost model. Next, we present in Section 3 the basic tabu search metaheuristic and the main features of the approach proposed in this paper: the definitions of solutions, moves, and neighborhood, as ingredients of the search strategy, and the implementation of the search memory through the use of a hashing-based data structure. Section 4 presents the computational results of our experiments, while some conclusions are drawn in the last section, concerning both the efficiency of tabu search for distributed query optimization and a discussion of the main features of the model and implementation developed in this work.

2 Query Optimization

2.1 Problem Description

Relational query optimization refers to the process of building an optimal execution plan, given an input query. Optimality is with respect to a cost function, used to compare the various solutions to the problem.

A *tuple* is a list of elements, called *attributes*. The number of attributes of a tuple is called its *arity*. The set of attributes of a tuple is called its *scheme*. A *relation* is a set of elements, each of which is a tuple with the same arity. Suppose we have two relations R and S, with sets of attributes $\{A_1, \ldots, A_m\}$ and $\{B_1, \ldots, B_p\}$, respectively. The *join* of R and S with respect to attributes A_i and B_j, respectively, is formed by taking each tuple r from R and each tuple s from S and comparing them. If the attribute A_i of r equals the attribute B_j of s, then a tuple is formed from r and s by taking the attributes of r followed by all attributes of s, omitting the repeated attribute B_j (see e.g. Aho and Ullman [1]). The scheme of the joined relation is then $\{A_1, \ldots, A_m, B_1, \ldots, B_{j-1}, B_{j+1}, \ldots, B_p\}$. A relational query language is a non-procedural one, i.e., a query specifies the wanted data and not the way the data is obtained. The following is a typical relational query:

Select $R_2.*$ **from** R_1, R_2, R_3, R_4
where $R_1.A = R_2.A$ **and** $R_2.B = R_3.B$ **and** $R_3.C = R_4.C$

This query involves three joins between four relations. The *query graph* depicted in Figure 1 is a subgraph of the database schema and establishes the join predicates (edges) between the relations (nodes) involved in the query. Here, we want to select all tuples from R_2 such that their A attribute is equal to the A attribute of a tuple from R_1, and the B attribute of the former is also equal to the B attribute of a tuple from R_3 whose C attribute is equal to the C attribute of a tuple from R_4. The join is an expensive operation, because it requires the matching of tuples of relations according to their join attributes. The equal comparison characterizes the *equijoin*, the most frequent, due to the need of normalizing relations, inherent to the relational model.

The first task of the query optimizer in a relational context is to decide in which order the relations are accessed. The *join ordering* problem is NP-hard if the number of relations of the

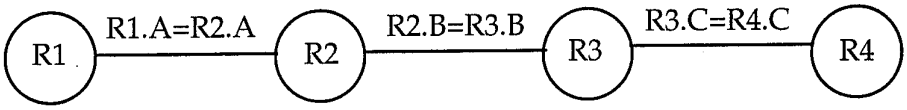


Figure 1: Query graph

input query is not fixed [15]. Furthermore, the optimizer must choose the *access strategy* for each individual relation. An access strategy involves, for example, choosing an access path, e.g. sequential scan or index. In a distributed environment, the relations may be located at different servers. Furthermore, relations may be partitioned among several servers, according to a range criteria. To perform a join between two relations, they must be located at the same site, or at least their corresponding fragments (when two relations are partitioned on the same servers and the same criteria is applied to the join attributes, the join may be concurrently executed on several servers and the partial results unioned and sent to the demander). In this case, the access strategy must also specify the location in which the join takes place. Either both relations or their corresponding fragments are located on the same site and the join may be performed without any redistribution, or a redistribution must be performed prior to the join.

An *execution plan* captures the join ordering, the access strategy for each individual relation and additional execution information. It is typically represented as a binary *operator tree*, where the leaves are base relations and the internal nodes are join (or union) operator nodes, whose result is captured by *transient relations*. The transient relations may or may not be materialized. Unary relational operators, such as selections and projections, are not represented, because they are applied "on-the-fly", before the tuple is passed to the join or union operation. Figure 2 shows two different execution plans for the sample query. Although not represented for simplicity, each join node captures a join algorithm (i.e., nested loop, merge join, or hash join) and an indication of redistribution, when needed. In our example, depicted in Figure 2, relations R_1 and R_2 are located at server S_1, while relations R_3 and R_4 are located at server S_2, i.e., the relations are pairwise located at different servers. Thus, both joins j_4 and j_5 can be executed respectively at servers S_1 and S_2 without any redistribution. However, join j_6 needs a redistribution prior to the operation: either the result of j_4 is sent to S_2 or the result of j_5 to S_1.

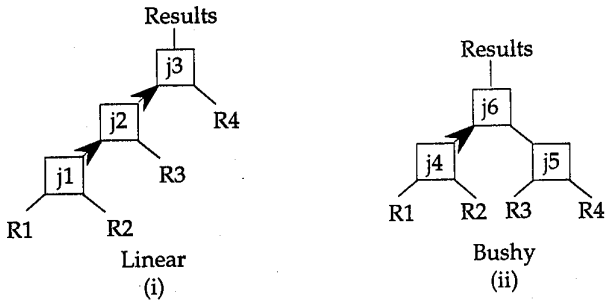


Figure 2: Execution plans as operator trees

Directed arcs linking two operator nodes express the fact that the transient relation produced by the first is consumed in *pipeline* by the second, i.e., in a tuple-by-tuple basis. Otherwise, the transient relation is completely *stored*, before it is consumed. Pipeline is obviously less expensive, and is chosen whenever possible. For example, all joins in Figure 2.(i) are executed using the nested loop algorithm. Thus, there is no need for materializing intermediate results. On the other hand, some join algorithms, e.g. merge join, require both relations to be sorted according to their join attributes. In this case, a previous materialization and sorting of an operand relation may be required.

We say that an operator tree corresponds to a *complete* execution plan if it captures all the relations of the input query (the two trees in Figure 2 represent complete plans). Otherwise, we say that the plan is *partial.*

2.2 Execution Space

The space of distributed execution plans is characterized by the nature of the investigated operator trees. A *linear space* is restricted to linear operator trees, where each join node has at least one base relation as one of its operands (see Figure 2.(i)). On the other hand, in a *bushy space*, a join node may have two transient relations as its operands (see Figure 2.(ii)). The bushy space is a superset of the linear space. If an exhaustive search strategy is used, the optimization cost is higher in bushy spaces, because it contains much more solutions than the linear one.

The choice of the execution space has a significant impact on the complexity of the query optimization problem. We now investigate the size of the search space. Let n be the number of relations in a query. Every operator tree associated with an execution plan has exactly n leaves. As an execution plan starts by joining two base relations and creates transient relations which are then one-at-a-time joined with another (either base or transient) relation, there are exactly $n-1$ internal nodes (corresponding to the join operations) in any operator tree. Supposing that the query graph is a clique (i.e., there is a join predicate between every pair of relations), then the number of possible operator trees is given by the solution of the recurrence equation

$$T_n = \sum_{k=1}^{n} \binom{n}{k} T_k T_{n-k},$$

with $T_0 = 1$. Its solution is given by

$$T_n = b_{n-1} \cdot n! = \frac{(2n-2)!}{(n-1)!},$$

where

$$b_{n-1} = \frac{1}{n} \binom{2(n-1)}{n-1}$$

is the $(n-1)$-th Catalan number. In fact, $T_n = (2n-2)!/(n-1)!$ is an upper bound for the size of the search space, as far as very often some relations are not connected by join predicates, or the search is restrained to linear execution plans (instead of general binary trees).

Typically, relational optimizers cut off the optimization cost by restricting the execution space to linear trees [22]. The use of dynamic programming is feasible in linear spaces for queries with moderate sizes (i.e., up to seven or eight relations) and guarantees that the best linear tree solution is found. However, in distributed environments, considering only linear operator trees is not an acceptable heuristic, as showed in [20]. In that work, it is demonstrated that in situations analogous to that of the example, where the relations are pairwise located at distinct servers, the optimal solution is always a bushy one, like the one in Figure 2.(ii), because it allows the joins j_4 and j_5 to execute concurrently. Only a few authors have considered the query optimization problem in bushy spaces and not too many algorithms and results are available [16, 20].

Another typical heuristic to cut off the space of solutions consists in investigating only operator trees that do not introduce unnecessary Cartesian products. For example, a join between R_1 and R_4 is a Cartesian product, since there is no join predicate that applies to it. The cardinality of the resulting transient relation would be the product of the cardinalities of the two relations. On the contrary, the cardinality of e.g. the resulting relation of the join operation j_4 is the product of the cardinalities of R_1 and R_2 reduced by a factor that reflects the *selectivity* of the join predicate $R_1.A = R_2.A$ (see [22] for selectivity estimates for several types of predicates). When avoiding Cartesian products, the only solutions investigated are such that there is always a join predicate applied at each node of the operator tree. This prevents generating large transient relations, and thus contributes to reducing the overall execution cost (or response time). However, in a distributed environment, this heuristic is not acceptable. As the communication costs impact significantly the overall cost, if the relations are located at the same site, performing the Cartesian product prior to redistributing may be worth doing from the point of view of execution cost. Thus, solutions containing intermediate Cartesian products should also be investigated.

Following this discussion, we have decided to focus our attention into the more difficult instances of the distributed query optimization problem, in terms of the query execution space. Thus, we consider query execution spaces in which bushy execution plans and Cartesian products are accepted, which are not dealt with very often in the literature.

To explore the space of solutions, either the optimizer builds operator trees or modifies complete trees, according to its *search strategy*. Strategies such as greedy or dynamic programming proceed by *building* plans, starting from base relations and joining one more relation at each step until complete plans are obtained. A greedy strategy builds only one such plan by depth-first search, while dynamic programming builds all possible plans by breadth-first search and selects the optimal one.

Local search optimization strategies, such as tabu search, concentrate on searching the best solution in the neighborhood of some particular one, without guaranteeing that the overall optimal solution is found. First, one or more *starting* plans are built by a greedy strategy. Then, the algorithm tries to improve the starting plan by visiting its *neighbors*. A neighbor is obtained by applying a *transformation* to a plan. Examples of typical transformations are shown in Figure 4. The optimizer search strategy is responsible for deciding how many points of the solution space are investigated and in which order. The choices are based on a cost function described in the next section.

2.3 Cost Model

The cost model gives a cost estimate for each operator tree. The cost refers to resource consumption, either space or time. Typically, query optimizers estimate the cost as time consumption. In a distributed execution environment, there are two different time consumption estimates to be considered: total time or response time. The former is the sum of the time consumed by each processor, regardless of concurrency, while in the latter it is considered that operations may be carried out concurrently. Thus, response time is a more appropriate estimate, since it corresponds to the time the user has to wait for an answer to the query.

In a distributed environment, the execution of an operator tree s is split into several phases. Pipelined operations are executed in the same phase, whereas a storing indication establishes the boundary between one phase and the subsequent one. For example, the operator tree in Figure 2.(i) is executed in one single phase, while the one in Figure 2.(ii) in two phases. Let $\Phi(s)$ denote the set of phases of plan s, while $\phi \in \Phi(s)$ denotes one individual phase. Moreover, let $O(\phi)$ denote the set of operations (i.e., the set of join operator nodes within the corresponding phase ϕ of the operator tree). Since the operations within the same phase are carried in parallel, the response time $respTime(\phi)$ of each phase $\phi \in \Phi(s)$ is the time needed to execute the most expensive operation, plus the necessary delay to start the pipeline, given by:

$$respTime(\phi) = max_{i \in O(\phi)}\{execTime(i) + pipeDelay(i)\},$$

where $i \in O(\phi)$ denotes an operator node, $execTime(i)$ is the execution time of operation i, and $pipeDelay(i)$ is the waiting period of operation i (time needed for the producer to deliver the first result tuples). The latter is null if the input relations of operation i are stored.

The cost $execTime(i)$ of an operator node i is build up from two components: the time to execute an operation and the the time to redistribute transient results (transmission time). The time to execute an operation depends on the chosen algorithm, e.g. nested loop for join. To compute it, the optimizer uses statistics stored on a metabase, that are periodically updated by the database administrator. Examples of such statistics are the cardinalities of relations and the number of distinct values of an attribute [22]. As an example of a cost function, we construct below the cost function $execTime(j_2)$ associated with the join node j_2 in Figure 2.(i). It consumes its left operand in pipeline, while the right operand is stored. Thus, the cost is:

$$execTime(j_2) = transmit(R_3) + max(cost_{nestedLoop}(join(j_1, R_3)), transmit(j_1)),$$

where $transmit(R)$ refers to the cost of redistributing R if needed. In a distributed environment, it is crucial to minimize this transmission time, since it impacts considerably the response time. Thus, our optimizer tries to exploit the placement of the operands to avoid redistribution. The transmission cost is estimated using system-dependent parameters, such as the network speed. In the current example, either j_1 or R_3 must be redistributed, since they are located at different servers. Thus, in the formula above one of the two transmission costs is different of zero and the other is null. The cost of computing the join of node j_1 with relation R_3 using, for example, the nested loop algorithm is:

$$cost_{nestedLoop}(join(j_1, R_3)) = card(j_1) * accessCost(j_1) + card(j_1) * card(R_3) * accessCost(R_3)$$

where, for any relation R, $accessCost(R)$ is equal to zero if relation R is not stored (i.e., it resides in the main memory) and $card(R)$ denotes the number of tuples in relation R.

To estimate the cost $f(s)$ of an operator tree s, the optimizer sums up the costs of all phases $\phi \in \Phi(s)$, which gives the total response time for executing the operator tree s:

$$f(s) = \sum_{\phi \in \Phi(s)} [respTime(\phi) + storeDelay(\phi)],$$

where $storeDelay(\phi)$ is the time necessary to store the output results produced by phase ϕ. The latter is null if ϕ is the last phase, assuming that the results are delivered as soon as they are produced. A more detailed description of the cost model may be found in Zaït [27].

3 Tabu Search with Hashing

Tabu search is an adaptive procedure for solving combinatorial optimization problems, which guides a hill-descending heuristic to continue exploration without becoming confounded by an absence of improving moves, and without falling back into a local optimum from which it previously emerged [6, 7, 8, 9, 10, 14]. Succesfull applications of tabu search for combinatorial problems have been reported in the litterature, see e.g. [2, 3, 5, 6, 11, 12, 13, 14, 21, 23, 25] among many other references cited e.g. by Glover [7] and Glover and Laguna [9]. A detailed description of the tabu search metaheuristic, together with other advanced features, improvements, extensions, and implementation strategies may be found in Glover and Laguna [9].

3.1 Algorithm Overview

Briefly, the tabu search metaheuristic may be described as follows. As a local search strategy, it starts by choosing an initial solution. At every iteration, an admissible move is applied to the current solution, transforming it into its neighbor with the smallest cost. Contrarily to a hill-descending scheme, one of the main features of tabu search is that, under certain conditions, moves towards a new solution that deteriorates the cost function are permitted at every local optimum. To avoid cycling by falling back into a local optimum from which the search procedure previously emerged, moves towards solutions already visited are interdicted. These restrictions have been implemented so far in almost all applications of tabu search by means of a short-term memory function which prohibits move reversals along some iterations. *Reversing moves* are those which would reverse some characteristics of the current solution, restablishing the values characterizing the solutions where the algorithm took non-improving moves. The number of iterations along which a move is prohibited is called its *tabu tenure*. Among other parameters and depending on the characteristics of each implementation, the performance of any algorithm using the tabu search metaheuristic is also intimately dependent on the maximum number of iterations, *maxmoves*, during which there may be no improvement in the best solution.

In this work, we use a different strategy to keep track of the memory function. Instead of prohibiting the reversion of characteristics which would lead back to solutions previously visited, we keep the whole list of all solutions already visited, examined, and explored during the search procedure. Now, an *admissible move* from the current solution is one that leads to some solution

not already visited during the search. The structure of the solutions, the dimension of the search space and the number of solutions which the algorithm must visit in order to find a sufficiently good, suboptimal solution make it possible for our implementation to store the whole list of solutions visited, through the use of a hashing-based data structure organized as described in Section 3.3 below. Figure 3 gives a procedural description of the basic tabu search metaheuristic.

```
begin
   Initialize the set of visited solutions with the empty set
   Generate the starting solution s_0
   s, s* ← s_0
   while (number of moves without improvement < maxmoves) do
   begin
      bestmovevalue ← ∞
      for (all candidate moves) do
      begin
         if (candidate move is admissible, i.e., if it does not lead to a solution already visited) then
         begin
            Obtain the neighbor solution s̄ by applying candidate move to the current solution s
            movevalue ← f(s̄) − f(s)
            if (movevalue < bestmovevalue) then
            begin
               bestmovevalue ← movevalue
               s' ← s̄
            end
         end
      end
      Insert s' in the set of solutions already visited
      if (f(s') < f(s*)) then s* ← s'
      s ← s'
   end
end
```

Figure 3: Description of the basic tabu search metaheuristic

3.2 Solutions and Moves: Exploring the Solution Space

In the case of query optimization, a *solution* is an operator tree whose nature is conform to the chosen execution space (see Section 2.2). The initial solution for tabu search is generated by a depth-first search greedy heuristic, e.g. the *augmentation heuristic* [24]. This heuristic chooses the relation with the smallest cardinality to start with. Then, it proceeds by joining a base relation at each step. Among all relations that might be joined to the current partial subtree, the heuristic keeps that leading to a new least costly extended subtree.

Tabu search builds one or more initial solutions and attempts to improve them by applying transformations (or moves) which lead to neighbor solutions of the current one. Only *valid transformations* that produce another complete tree in the same solution space are applied. For example, if the search space is such that Cartesian products are not investigated, then a valid transformation must not produce a tree with a Cartesian product: a tree like join(join(join(R_1,R_4),R_2),R_3) would be invalid in our original example. The set of available transformations is also conditioned by the shape of the investigated operator trees:

- In a *linear space*, a transformation is a *swap*, which chooses randomly two nodes in the current tree and swaps the relations consumed by them (see Swami [24] and Figure 4.(i)).
- In a *bushy space*, valid transformations are the *swap* (as above), the *join commutativity* (see Figure 4.(ii)) and the *join associativity* (join(join(R_i,R_j),R_k) → join(R_i,join(R_j,R_k)), see Figure 4.(iii)).

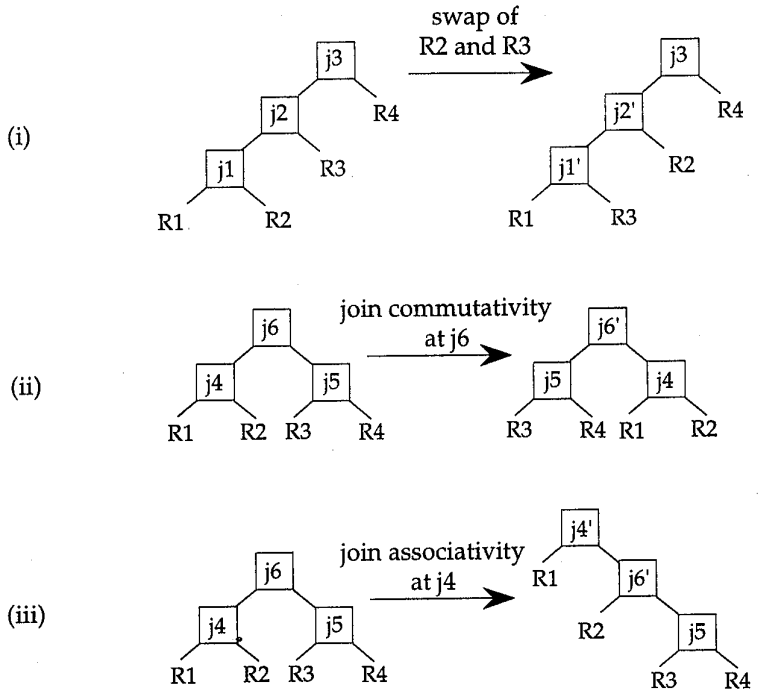


Figure 4: Actions for transforming an operator tree

We implement operator trees using a recursive tree structure. If the optimizer were to work only with linear trees, we could have chosen to represent linear trees as a permutation of integers (as for many other scheduling problems in combinatorial optimization), each integer refering to a base relation. Manipulating non-linear trees adds some complexity to the implementation of moves, and makes it more difficult to exhaust the whole set of possible moves. There are $O(n^2)$ transformations associated with each operator tree.

The implementation of the swap transformation is the same for linear or bushy spaces. The swapping nodes are systematically chosen by means of two nested loops searching through all pairs of nodes. For each pair of relations exchanged, a new neighbor solution is generated and examined. In bushy spaces, besides the swaps, each node of the current solution is also submitted to join commutativity transformations and to join associativity transformations, which are always local transformations rooted at one node of the tree. The neighbor solutions are tested for their validity from the point of view of Cartesian products.

In the case of linear spaces, this implementation guarantees that every solution may be attained from any initial solution by an appropriate sequence of swap transformations. However, in the case of bushy spaces, the only described swap, join commutativity and join associativity

transformations are not able to exhaustively generate the whole execution space. Although a more time-consuming procedure would be necessary for giving to the algorithm the capability of investigating the whole solution space by exhaustively generating complete neighborhoods, the computational results presented in Section 4 illustrate that the restricted neigborhoods together with the search scheme are already able to provide sufficiently good results.

The algorithm incorporates some diversification features, by restarting the search from another initial operator tree whenever all moves from the current solution are tabu or a given maximum number of iterations has been performed without improvement in the best solution found so far. The new starting solution is also obtained by the greedy augmentation heuristic. Now, the latter privilegiates the beginning of the construction of the operator tree from a new initial base relation, which was not yet used as the first one during the construction of the previous starting solutions.

3.3 Implementing the Search Memory by a Hashing-Based Structure

Contrarily to randomized algorithms, which have no memory of the search path, tabu search uses a short term memory function to keep track of previously investigated solutions (or, more often, of their characteristics or attributes). A crucial problem when implementing tabu search is choosing the structure that implements the search memory. Since each solution is represented by a long data structure, storing all solutions visited throughout the search is usually unfeasible, both in terms of the space to store them and the time to compare a new solution with all other previously visited. One solution adopted in precursor tabu search implementations was to keep a circular list with the most recently visited solutions. Since not all visited solutions are stored, this approach may lead to cycles longer than the size of the circular list. A second solution consists in memorizing a list of *tabu-active* solution characteristics (or move characteristics) which may not be reversed during a certain number of iterations, instead of the solutions themselves. For example, in the case of swap moves, a characteristic may refer to the position of the two swapped nodes. The disadvantage of this approach is that it may lead to very restrictive search paths prohibiting the search to visit solutions not yet investigated, only because they show some characteristic whose reversal may be ambiguously interdicted.

In the sequel we show that, in the case of the distributed query optimization problem, the search memory can store the whole set of visited solutions, or some unique representation of each of them. We use a hashing-based data structure for storing the entire history of the search, i.e., all solutions visited by the algorithm. The use of hashing vectors for tabu search has been previously discussed in [26], although no specific application was either given or evaluated in details in computational terms.

To implement the hashing-based data structure, each solution is first *flattened* to a string by scanning the operator tree in pre-order. Thus, a solution like the one in Figure 2.(ii) becomes the string **12*34, where * stands for visiting an internal node of the operator tree, corresponding to the root of a subtree. This provides a unique representation for each solution. This flattened representation enables a fast comparison of solutions, in a byte-per-byte basis.

The search memory for storing the search path is composed by an array of short lists. The array is indexed by a key obtained by a hashing function applied to the flattened representation of each visited solution. Each element of the array points to a list containing all visited solutions associated with the same key. The key associated with an operator plan s is generated by using a hashing function, such as those proposed in [26]:

$$h(s) = \left[\sum_{i=1}^{n} n^{i-1} s_i\right] \bmod(\texttt{MAXINT} + 1),$$

where n is the number of relations in the query, s_i is the i-th relation sequence number in the flattened representation of the operator plan s (regardless of the positions storing an *), and MAXINT is the maximum integer value which may be stored using two bytes, i.e. 65,535. Thus, the key for the tree in Figure 2.(ii) is $4^0 \cdot 4 + 4^1 \cdot 3 + 4^2 \cdot 2 + 4^3 \cdot 1 \bmod(65,536)$, giving an integer in the range of 0 to 65,535, which is the size of the array.

Notice that all solutions associated with the same sequence of relations (when the leaves of different operator trees appear in the same order, regardless of the topology of the tree) will have the same key. Thus, both trees in Figure 2 have the same key, although their flattened representations are different. The collision probability with this hashing function has been previously studied in [26], where it is shown to be low. In our case, it is slightly higher, due to the fact that the topology of the tree is not considered.

We briefly discuss the impact of such strategy in terms of space and time consumed by the algorithm. The array of pointers consumes 256 Kbytes for storing 65,536 elements (each element stores a 4-byte pointer), while each solution needs $2n - 1$ characters to be stored, where n is the number of relations in the query. Even when the number of visited solutions is very high, e.g., 10,000, the search memory requires only additional 190 Kbytes in the case of a query with 10 relations ($10,000$ solutions $\times$ 19 bytes per stored solution).

4 Applications and Computational Experiments

In this section, we present some experimental computational results obtained through the use of our tabu search approach. We consider a real-life application, where queries range from 4 to 12 relations. In a distributed execution environment, such queries make a dynamic-programming-based optimizer run out of time and space, because of the complexity incurred by distribution, leading to the need for approximate, but faster, strategies. We show that tabu search finds better solutions than simulated annealing which, in turn, was already shown to perform better than hill-descending [20, 24]. All these strategies are available as options of the query optimizer used in our experiments, originally built for the EDS ESPRIT project [4, 18, 19, 20].

As already commented in Section 2.2, we have decided to focus our attention into the more difficult instances of the distributed query optimization problem, in terms of the query execution space, i.e., we consider query execution spaces in which bushy execution plans and Cartesian products are accepted.

4.1 Test Problems from a Real-life Application

Our computational experiments are based on a real-life application coming from PETROBRAS, the Brazilian state-owned petroleum company in charge of oil prospection, exploitation and distribution. The application refers to geological research for locating new wells. For this purpose, the company acquires, process and analyses seismical data. There are many seismical lines, which

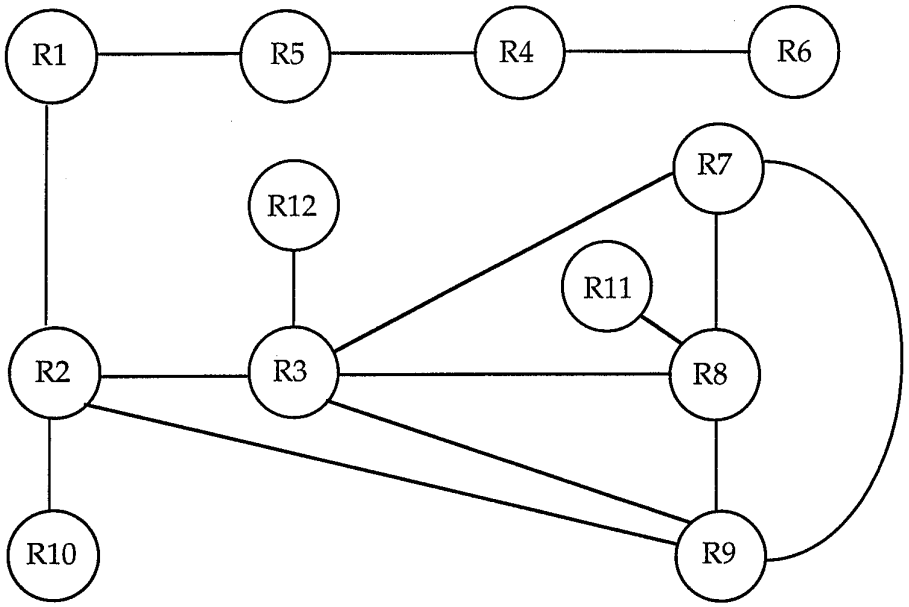


Figure 5: Database schema

are grouped into blocks to ease their computational treatment. The application also controls the work of the technicians in charge of the analysis of seismical data. Brazil being a large country with many areas under exploration, all this represent a large volume of information.

We represent the database as a graph, where a node corresponds to a relation and an edge connects two nodes if the corresponding relations share the same attribute name. The database schema in Figure 5 contains 12 relations joinable through foreign keys. There are three kernel relations: "seismic-lines" (R_2), "line-reprocessing" (R_3) and "reprocessing-detail" (R_9), to which the remaining relations are joinable. The query graph representing some query is a subgraph of the database schema. We conducted our experiments using equijoin queries ranging from 4 to 12 relations. The relations involved in each test problem are grouped together as described in Table 1. Each query was optimized supposing that the relations were distributed over up to four sites. Table 2 gives the values for some parameters of the cost model.

The tabu search algorithm was implemented in $C++$. The computational results are reported in the next sections. The execution costs or response times refer to the cost function $f(s)$, which gives the time needed to provide the results for a given query when the operator tree s is used. The optimization cost stands for the computational time required by the query optimizer. The computational experiments with the optimizer were performed on a Sun SPARCstation-2 workstation with a 16 MIPS processor and 32 Mbytes of memory.

4.2 Optimization Cost and Execution Space

A common difficulty associated with the measurement of resource consumption is that it is very implementation-dependent. As far as the processing time in a multitasking environment is not a very precise measure for evaluating the behavior of some algorithm, we have used another machine-

Number of relations	Relations appearing in the query
4	R_1, R_4, R_5, R_6
5	all the previous, plus R_2
6	all the previous, plus R_9
7	all the previous, plus R_7
8	all the previous, plus R_8
9	all the previous, plus R_3
10	all the previous, plus R_{10}
11	all the previous, plus R_{11}
12	all the previous, plus R_{12}

Table 1: Relations appearing in each test problem

Parameter	Value
Number of MIPS per processor	16 MIPS
Speed of the network	1.25 Mbytes/second
Size of a packet	512 bytes
Time for a send operation	1000 instructions or 33 μs
Time for a receive operation	700 instructions or 23 μs

Table 2: Parameter values for the cost model

independent measure for the optimization cost.

First, we remark that the neighborhood search and the computation of the response time through the cost function $f(.)$ described in Section 2.3 are the most computation-intensive steps of our tabu search algorithm. Whenever a move leads from the current execution plan A to one of its neighbors B, we start building this new plan B from the largest subtree of A entirely contained in it. From this point on, new nodes are created corresponding to the join operations on the upper part of the operator tree associated with B. The response time $f(B)$ is also progressively recomputed from the value of the execution time at the root of the above mentioned subtree, until the construction of B is completed. Then, the generation of a new node involves the allocation of a new memory position, setting a pointer from this new position to the address of a subtree, and computing the response time associated with the corresponding operator.

We propose to measure of the optimization cost by the overall number of new nodes which are created and investigated during all neighborhood searches and at each iteration when the best non-tabu move is effectively performed. This number of generated nodes provides an implementation-independent measure to evaluate the amount of resources consumed by the optimizer, i.e., the optimization cost.

Figure 6 illustrates the computation of the number of generated nodes through a small example. Figure 6(i) shows a query graph involving four relations. An execution plan A is shown in Figure 6(ii). A move applied to A by swapping relations R_3 and R_4 and leading to the execution plan B^1 in Figure 6(iii) accounts for two generated nodes, as far as the operator node j_1 is kept and two new nodes j_4 and j_5 are created. Another move applied to A by swapping relations R_2

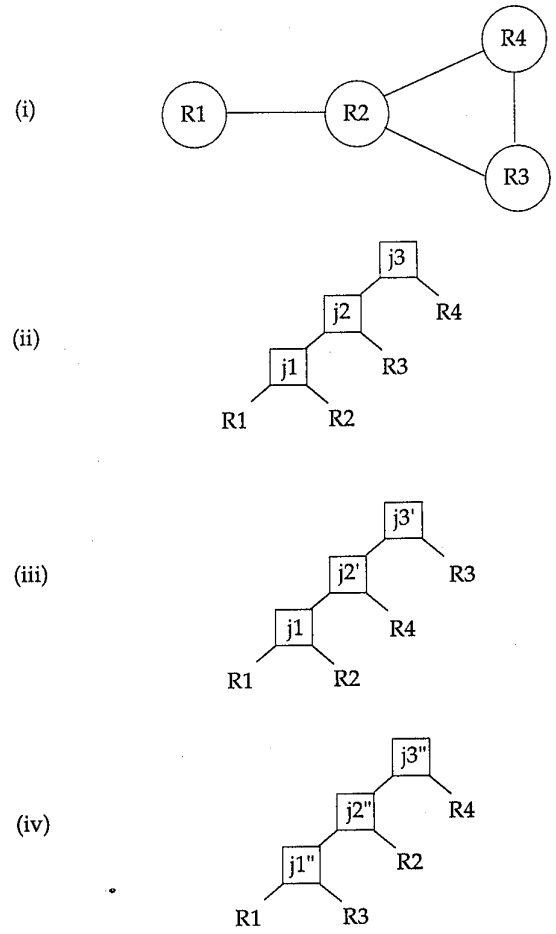


Figure 6: Computation of the number of generated nodes

and R_3 and leading to the execution plan B^2 in Figure 6(iv) accounts for three generated nodes. Notice that if Cartesian products were not allowed, the evaluation of the latter would account for only one generated node, as far as the construction of B^2 would be immediately interrupted following the identification of a Cartesian product at node j_6.

We chose the bushy execution space for the computational experiments, as it is has been shown in Section 2.2 to be the most appropriate for distributed queries. Cartesian products are also accepted, leading to more difficult instances of the distributed query optimization problem, for which not too many results are available in the literature.

Dynamic programming was applied to problems involving up to seven relations, distributed over up to three sites, considering a bushy execution space where Cartesian products are accepted. Figure 7 illustrates the improvement in the optimal response time with the increase in the number of sites, showing that distributing the relations of the database over many sites leads to smaller response times. The larger is the number of sites, the larger is the level of concurrency among the servers, which contributes to the reduction of the overall response time (although there will be a threshold on the number of sites, beyond which the increase in the transmission costs will neutralize the speedup obtained with concurrency). On the other hand, a larger number of sites increases the

possibilities of execution, due to the existence of more alternatives of redistribution. Figure 8 shows that increasing the number of sites makes the optimization cost of dynamic programming grow drastically higher. In fact, dynamic programming is useless even for medium-size queries, as far as it runs out of time and space. This fact clearly establishes the need for good approximate algorithms, which may find a good solution in terms of response time under a reasonable optimization cost. The computational results obtained by tabu search are presented in the next section.

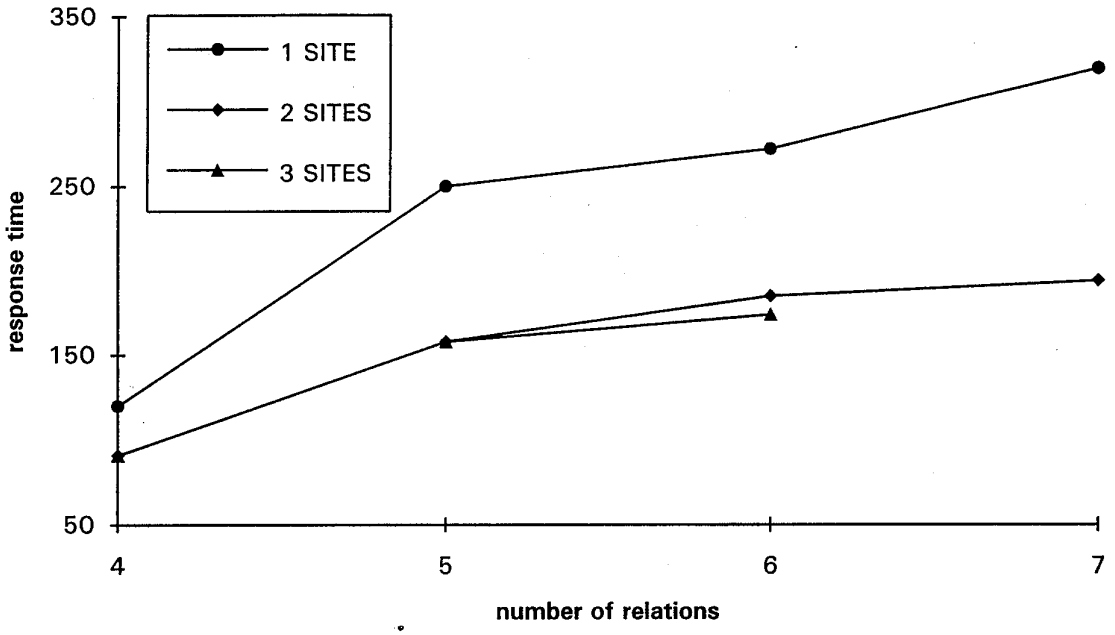


Figure 7: Response time vs. number of relations (bushy space with Cartesian products)

4.3 Solution Quality vs. Search Strategy

We applied three different strategies (simulated annealing, dynamic programming, and tabu search) to several queries involving up to 12 relations in a bushy space with Cartesian products, both in a centralized execution environment (one site) and in a distributed environment (2, 3, and 4 sites). We notice that dynamic programming runs out of time and space for problems with more than six relations and two sites.

Figure 9 shows the results obtained for one site (centralized case) and four sites (distributed case). For all small problems, both in the centralized as well as in the distributed case, tabu search and simulated annealing found the same exact optimal solution obtained by dynamic programming. For all problems, tabu search always obtained solutions at least as good as those found by simulated annealing, and improving the latter by as much as 18% in terms of the response time.

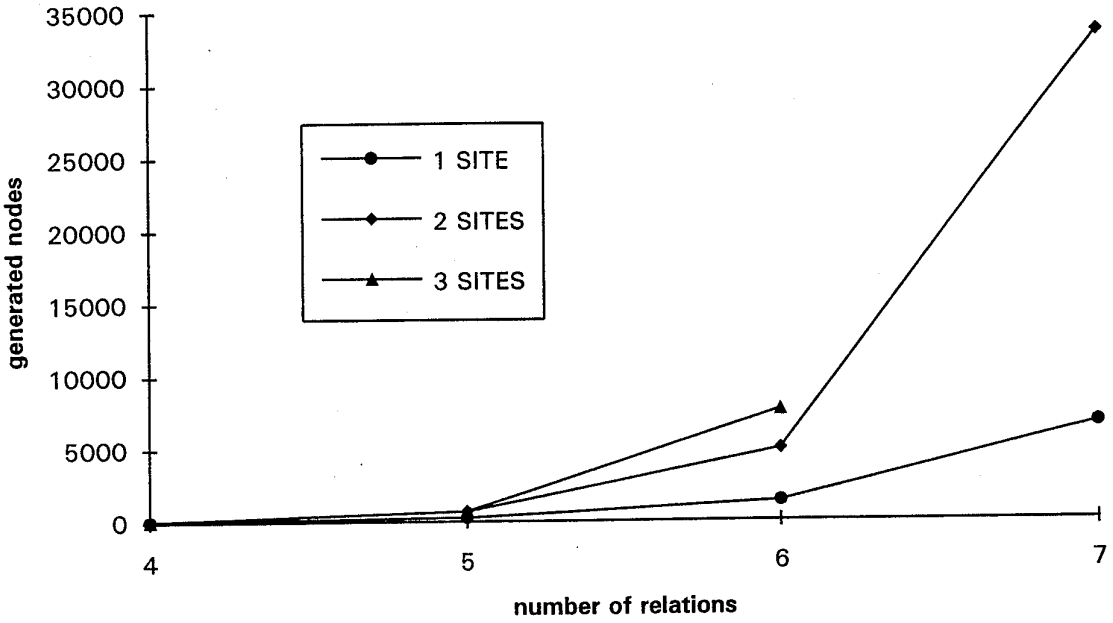


Figure 8: Optimization cost vs. number of relations (bushy space with Cartesian products)

5 Conclusions

In this paper, we considered the query optimization problem in distributed relational databases. We showed that the complexity of the problem increases with the distribution of the relations over several sites. Good heuristics for centralized environments do not apply to the distributed case, due to the need for accepting Cartesian products and more complex bushy operator trees than the linear execution plans usually considered in the literature. Moreover, dynamic programming runs out of time and space even for small problems with as few as seven relations.

We have then proposed the application of tabu search to distributed query optimization. Our implementation of tabu search has been significantly simplified through the use of a search memory which keeps track of the whole set of visited solutions, by means of a hashing-based data structure for storing the entire history of the search.

We performed several computational experiments based on a real-life application coming from PETROBRAS, the Brazilian state-owned petroleum company in charge of oil prospection, exploitation and distribution. The test problems involve as many as 12 relations, distributed over up to four sites. The effectiveness of tabu search for distributed query optimization was clearly established. It found optimal solutions for all small- and medium-size problems, and much better solutions than other local search strategies, such as simulated annealing, for large-size problems. As a consequence, tabu search is now being offered as one of the search strategies implemented in the query optimizer.

To conclude, we may say that the approach proposed in this paper is one of the first successfull attempts to solve the more difficult instances of the distributed query optimization problem, in which bushy execution plans and Cartesian products are accepted.

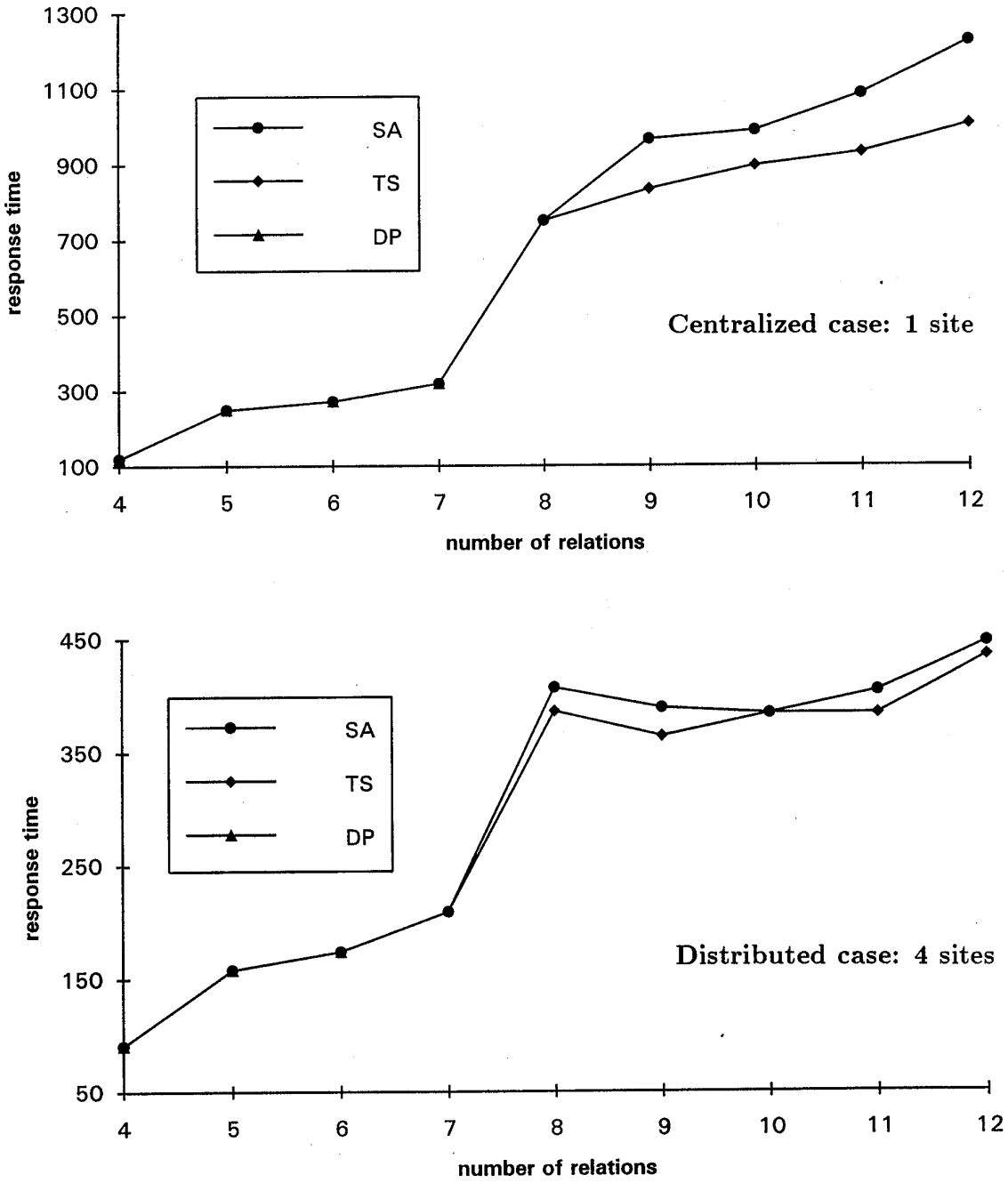


Figure 9: Results obtained by tabu search (bushy space with Cartesian products)

References

[1] A.V. AHO and J.D. ULLMAN, *Foundations of Computer Science*, W.H. Freeman and Company, New York, 1992.

[2] S.G. DE AMORIM, J.-P. BARTHELÉMY, and C.C. RIBEIRO, "Clustering and Clique Partitioning: Simulated Anealing and Tabu Search Approaches", *Journal of Classification* 9 (1992), 17–41.

[3] A.A. ANDREATTA and C.C. RIBEIRO, "A Graph Partitioning Heuristic for the Parallel Pseudo-Exhaustive Logical Test of VLSI Combinational Circuits", *Annals of Operations Research* 50 (1994), 1–36.

[4] EDS DATABASE GROUP: "EDS-Collaborating for a High-Performance Parallel Relational Database", *ESPRIT Conference, Brussels*, 1990.

[5] C. FRIDEN, A. HERTZ and D. DE WERRA, "STABULUS: A Technique for Finding Stable Sets in Large Graphs with Tabu Search", *Computing* 42 (1989), 35–44.

[6] F. GLOVER, "Tabu Search – Part I", *ORSA Journal on Computing* 1 (1989), 190–206.

[7] F. GLOVER, "Tabu Search – Part II", *ORSA Journal on Computing* 2 (1990), 4–32.

[8] F. GLOVER, "Tabu Search: A Tutorial", *Interfaces* 20 (1990), 74–94.

[9] F. GLOVER and M. LAGUNA, "Tabu Search", Chapter 3 in *Modern Heuristic Techniques for Combinatorial Problems* (C.R. Reeves, Ed.), 70–150, Blackwell Scientific Publications, 1993.

[10] F. GLOVER, E. TAILLARD, and D. DE WERRA, "A User's Guide to Tabu Search", *Annals of Operations Research* 41 (1993), 3–28.

[11] P. HANSEN, E.L. PEDROSA FILHO, and C.C. RIBEIRO, "Location and Sizing of Off-Shore Platforms for Oil Exploration", *European Journal of Operational Research* 58 (1992), 202–214.

[12] P. HANSEN, M.V. POGGI DE ARAGÃO, and C.C. RIBEIRO, "Boolean Query Optimization and the 0-1 Hyperbolic Sum Problem", *Annals of Mathematics and Artificial Intelligence* 1 (1990), 97–109.

[13] A. HERTZ and D. DE WERRA, "Using Tabu Search Techniques for Graph Coloring", *Computing* 29 (1987), 345–351.

[14] A. HERTZ and D. DE WERRA, "The Tabu Search Metaheuristic: How We Used It", *Annals of Mathematics and Artificial Intelligence* 1 (1990), 111–121.

[15] T. IBARAKI and T. KAMEDA, "On the Optimal Nesting Order for Computing N-Relational Joins", *ACM Transactions on Data Bases* 9 (1984), 482–541.

[16] Y.E. IOANNIDIS and Y. KANG, "Left-deep vs. Bushy Trees: An Analysis of Strategy Spaces and Its Implications for Query Optimization", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1991.

[17] Y.E. IOANNIDIS and E. WONG, "Query Optimization by Simulated Annealing", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 9–22, San Francisco, 1987.

[18] R.S.G. LANZELOTTE and P. VALDURIEZ, "Extending the Search Strategy in a Query Optimizer", *Proceedings of the 17th International Conference on Very Large Data Bases*, 363–373, Barcelona, 1991.

[19] R.S.G. LANZELOTTE, P. VALDURIEZ, and M. ZAÏT, "Optimization of Object-Oriented Recursive Queries using Cost-Controlled Strategies", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 256–265, San Diego, 1992.

[20] R.S.G. LANZELOTTE, P. VALDURIEZ, and M. ZAÏT, "On the Effectiveness of Optimization Search Strategies", In *Proc. 19th Int. Conf. on Very Large Data Bases*, 493–504, Dublin, 1993.

[21] S.C. PORTO and C.C. RIBEIRO, "A Tabu Search Approach to Task Scheduling on Heterogeneous Processors under Precedence Constraints", *International Journal of High Speed Computing* 7 (1995), no. 2.

[22] P.G. SELINGER, M.M. ASTRAHAN, D.D. CHAMBERLIN, R.A. LORIE, and T.G. PRICE, "Access Path Selection in a Relational Data Base System", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 23–34, Boston, 1979.

[23] J. SKORIN-KAPOV, "Tabu Search Applied to the Quadratic Assignment Problem", *ORSA Journal on Computing* 2 (1990), 33–45.

[24] A. SWAMI, "Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 367–376, Portland, 1989.

[25] M. WIDMER and A. HERTZ, "A New Approach for Solving the Flow Shop Sequencing Problem", *European Journal of Operational Research* 41 (1989), 186–193.

[26] D.L. WOODRUFF and E. ZEMEL, "Hashing Vectors for Tabu Search", *Annals of Operations Research* 41 (1993), 123–137.

[27] M. ZAÏT, *Optimisation de Requêtes Relationnelles pour Exécution Parallèle*, Doctorate dissertation, Université de Paris VI, 1994.