Computers and Artificial Intelligence. Vol. 12. 1993. No. 2. 145-166

PARALLEL OPTIMISATION OF JOIN QUERIES USING A TECHNIQUE OF EXHAUSTIVE NATURE[1]

Maria SPILIOPOULOU, Michalis HATZOPOULOS, Costas VASSILAKIS[2]

Department of Informatics, University of Athens
Panepistimiopolis. TYPA Buildings, Ilisia, GR 157 71

Abstract. In this study *we* present a technique for the parallel optimisation of join queries that uses the offered coarse-grain parallelism of the underlying architecture in order to reduce the CPU-bound optimisation overhead. The optimisation technique performs an almost exhaustive search of the solution space for small join queries and gradually, as the number of joins increases, it diverges towards iterative improvement. This technique has been developed on a low-parallelism transputer-based architecture, where its behaviour is studied for the optimisation of queries with many tens of joins.

Keywords: parallel optimisation, join queries, coarse-grain parallelism, transputers.

1 INTRODUCTION

One of the problems recently addressed in database research is the optimisation of large join queries. Conventional optimisation models are based on exhaustive techniques, the overhead of which increases exponentially with the query size [13]. Therefore, researchers focus on techniques based on iterative improvement, like Swami and Gupta [13, 14]. or simulated annealing, like loannidis, Wong and Kang [2, 3], aiming at a polynomially increasing optimisation overhead.

Query optimisation techniques are CPU-intensive applications: so they may benefit from parallelism. Both exhaustive techniques, which construct all query execution plans to find the optimal one, and non-exhaustive techniques of iterative nature, like iterative improvement and simulated annealing, are clearly parallelisable.

In this study, we propose a query optimisation model, which uses parallelism not only for query execution hut also for query optimisation. We present a parallel optimisation technique, hereafter denote as the *hybrid technique,* to stress the fact that it is almost exhaustive for small join queries, and diverges gradually from exhaustive techniques for larger queries towards iterative improvement. The overhead of this technique is less than the overhead of an exhaustive strategy and is further reduced proportionally to the utilised processors.

The paper is structured as follows: In the next section, we make a brief overview of query optimisation techniques. In Section 3 we focus on parallel environments and outline the query processing model, for which the optimisation technique is designed. In Section 4, we

[1] An earlier version of this paper has been published in the PA RLE '92 Conference under the title "Using Parallelism and Pipeline for the Optimisation of Join Queries". pp. 279-294.

[2] Part of this work has been performed within the project 4061 Parallel Computing Action (PCA). partially funded by the CEC.

analyse the proposed hybrid technique, and we briefly outline an alternative method based on iterative improvement which has been used for performance comparisons. This method is hereafter denoted as *iterative-improvement-technique.*

Section 5 presents the first results of an experimental implementation of the model on a transputer machine, as well as the comparisons of the behaviour between the hybrid and the iterative-improvement techniques for queries containing tens of joins. The final section concludes the paper and presents some future extensions of the model.

2 OPTIMISATION TECHNIQUES FOR JOIN QUERIES

Query optimisation aims at the selection of an optimal access plan out of all plans that can be constructed by placing the query operators in ail possible orderings and assigning to them any of the execution algorithms available in the literature. Adopting the terminology of combinatorial optimisation problems, each query access plan can be observed as a *state* in the *space* containing all solutions to the problem. The *optimal state* is obviously a state of the solution space that has the lowermost cost.

In most query models, query optimisation is reduced to the optimisation of join operators: which in turn is reduced to finding the best join order and to assigning the most appropriate algorithm to each of the joins, as analysed by Ioannidis and Kang [4]. In this study, we also focus on the optimisation of join queries, although the impact of other operators *on* the access plans and, especially, on the optimisation process itself also deserves a thorough study, as mentioned elsewhere [9].

As pointed out by loannidis and Kang [4], a state in the solution space of the join query optimisation problem corresponds to the access plan for a "join processing tree" [4]. The join processing tree is a special case of a query tree, the nodes *of* which are query operators, while the vertices represent the flow of information from the leaves obtaining the relations from secondary storage towards the root producing the final output. The join processing tree is thus a query tree comprised solely of JOIN-nodes.

The exhaustive searching of the solution space in order to select the state w$_{it}$h the minimum cost has a rapidly increasing overhead, since the number of states grows exponentially to the number of operators in the query. Thus, the optimisation overhead reaches unacceptably high values, especially for large join queries, as mentioned by Swami and Gupta [13].

In the model of Selinger et al. [7] the optimisation overhead is reduced by the use of heuristics, which excludes a number of states from the solution space as potentially suboptimal. However, the combinatorial nature of the problem is not overcome: as recent research focuses on the optimisation of large join queries, the usage of optimisation techniques of exhaustive nature becomes inappropriate.

In order to optimise large join queries, researchers focus on "randomised algorithms", which are used in an iterative fashion, as described by loannidis and Kang [3]. The iterative techniques employed are based on iterative improvement, as in the models of Swami and Gupta [13] and Swami [14], on simulated annealing, as in the model of Ioannidis and Wong [2], and on combinations of the two, as in the model of loannidis and Kang [3, 4].

The models of Swami and Gupta [13, 14] use a special type of join processing tree the Outer Linear Join Tree (OLJT). The main characteristic of the OLJT is that it holds for each JOIN-node that the inner operand relation is always a base one. By using only the hash algorithm for all joins, the optimisation problem is further reduced to that of finding the optimal join ordering. To resolve this problem, the iterative improvement technique combined with augmentation heuristics is suggested as the most efficient approach [14].

The model of loannidis and Wong [2] uses simulated annealing, while the model of Ioannidis and Kang [3. 4] combines simulated annealing with iterative improvement for deep and bushy join processing trees, which are more general than the left-deep OLJTs. The study shows the superiority of the composite technique, called 2-phase optimisation, for all solution spaces but those where local minima are scattered at all costs [4].

The aforementioned approaches leave an intermediate area of solution spaces for deep and bushy trees, where local minima are scattered at all costs. Moreover, in both optimisation techniques, both the exhaustive ones and those based on randomised algorithms, the fact is overseen that the optimisation overhead can be reduced in much the same way as the execution cost of the queries, namely by using the available processing power of the parallel machines, on which the database models operate.

3 THE ENVIRONMENT OF THE PARALLEL OPTIMISER

The primary objective of most query optimisers is the minimisation of query execution time. Since database research has focused on the optimisation of large join queries, the minimisation of optimisation overhead becomes an additional objective for modern query processing models.

Our optimisation model utilises parallelism to minimise both the (usually I/O-bound) execution time and the CPU-bound optimisation overhead. The model is designed for coarse-grain parallelism architectures comprised by a small number (a few tens) of processors with private memories, point-to-point interprocessor communication and shared peripherals: however, the proposed technique can also be used on machines with private peripherals per processor. These architectures conform to the requirements of recent database machines, as identified in the overview of De Witt and Gray [1].

The model tranforms the initial query into the query tree representation we proposed in [11]. Since in this study we focus on the optimisation of join queries, we only consider join processing trees, hereafter also denoted as JTs. More specifically, our model is designed to process SQL queries with subqueries that may be nested in any depth. The queries can contain ail relational algebra operators: set operators (union. intersection. set difference, can also appear between subqueries. For the purposes of this study, we only focus on join queries produced using SQL: however, the same technique is applicable when all query operators are considered, as in the model we propose in [9].

The JTs handled by the optimiser are both deep and bushy, like the JTs addressed in the model of Ioannidis and Kang [4]. This tree structure has two advantages: (I) It reduces the optimisation problem to that of finding the optimal join order and the most appropriate algorithms for the nodes [4], as already noted, and (ii) it is very suitable for low parallelism architectures, since it can be directly mapped one-to-one into the query execution graph on the parallel machine: tree nodes are mapped into execution tasks, while tree vertices are implemented as pipes on the execution graph.

The optimal join tree for a query is a join tree with smallest cost, constructed by repeated tree reorganisation and algorithms selection. The optimiser uses a library of uniprocessor join algorithms, since the model runs on a low parallelism environment, where each tree node is executed by a single processor. The following join algorithms are available for all join operators, including the anti-join operator introduced by Kim [5]:

- the classic hash method for joins and outerjoins bearing the equality operator, as well as for anti-joins
- the nested loops method for joins on relations, one of which fits in a processor's main memory
- the merge method for joins on relations, all of which are sorted on the join attribute.

The execution algorithm assigned to a JOIN-node is not randomly selected, as in the model of loannidis and Kang [4], because the behaviour of the three aforementioned algorithms is well studied as in the work of Kim [5], Shapiro [8] and Mikkilineni and Su [6], so that criteria for their usage can be extracted:

1. The classic hash method shows best behaviour when the relation on which the hash table is constructed fits in a processor's memory, as proven by Shapiro [8]. The method can still be used even if only the hash table fits in memory, since this algorithm is proven to be superior to the other two methods in most cases. as noted by Swami and Gupta [13].
2. The nested loops algorithm performs optimally when the relation selected as inner one fits in memory. This algorithm has a further advantage of being applicable to all kinds of joins, irrespective of their comparison operator.
3. The merge algorithm require both input relations to be sorted on the join attribute. Whenever this can be ensured, this algorithm is the more appropriate one. Otherwise, it is inferior to the hash method. Its superiority or inferiority towards the nested loops method depends on whether the cost of sorting the input relations exceeds the cost of swapping one of them in and out of secondary storage, as described by Kim [5].

The algorithm selection for each node is performed according to these criteria.

4 TREE REORGANISATION IN .A PARALLEL ENVIRONMENT

In this section, we first analyse the sequential version of the proposed hybrid technique and then describe the usage of parallelism in its implementation. The iterative-improvement technique is outlined in-between, since the impact of parallelism on its behaviour is analogous to that of the hybrid one.

4.1 States and moves for tree reorganisation

Adopting the terminology used by Swami and Gupta [13] and by loannidis and Kang [4] for non-exhaustive optimisation techniques, we use the following terms:

The *initial stare* of the query optimisation technique is the JT input to the optimiser. On this tree a *set of moves* is applied, denoted as "set of transformation rules" in [4]. A move is called *legal* if it produces a state having lower execution cost than the state it was applied on. The states produced by applying a move to a state S define the *neighbourhood of* S and are called *neighbours of* S [4].

Local search starts by applying random moves on each of the *start states,* in order to produce a *local minimum* [13]. The start state is a state constructed by random placement *of* operators or by using augmentation heuristics. as in the model of Swami [14]. The local minimum with the lowermost cost among the computed ones is declared as the *global minimum* [13].

The moves performed by the hybrid and the iterative-improvement technique are based on:

* the *swap(Join1 Join2)* operation for transformations on a state
* the *reroot(State, Join0)* operation for the construction of start states.

The *swap()* operation shifts the positions *of* the nodes Join1 and Join2, which must be adjacent. swap() directly affects the position of all successors of the two nodes. but its impact on the execution plan spreads across the whole tree: the procedure assigning algorithms to the nodes is invoked for the new tree, thus constructing a new state-executjon pjan.

The arguments to the *reroot()* operation are the current state, which corresponds to a JT rooted at a JOIN-node JoinX, and a node Join() other than JoinX. This operation creates a new JT rooted at node Join0 and having all other nodes of the JT placed below Join0). The placement of nodes takes place using the mechanism we proposed in the pre-optimisation model of [11], which ensures that tree pipeline is not corrupted, i.e. that adjacent nodes are applied at least on one common relation. The result is an equivalent JT of different structure. As in the case of swap(), algorithms are assigned to all nodes of this tree anew, thus producing a new state.

In order to compute a state's cost, we developed a set of cost formulae which are estimating the I/O and communication cost required by each strategy for execution in a parallel environment. The formulae are based on the following guidelines: (a) tree nodes are executed as tasks in parallel and pipelined fashion. (b) tasks exchange data asynchronously, (c) all algorithms filter out attributes not used in ancestor nodes and (d) projections perform sorting and (occasionally) duplicates removal [10. 12].

4.2 Description of the hybrid technique

The hybrid technique creates N neighbours of the initial state S_{init} using the *reroot()* operation. Those neighbours are used as start states. Thus, *reroot( )* is actually a move, the legality of which is not tested.

On each start state S, a procedure SCAN( ) is applied, which repeatedly calls *swap()* to search the neighbourhood *of* S in an almost exhaustive way. SCAN() operates on a start state S, applying the *swap()* operation in a loop fashion as follows:

- The start state is marked as the current state. The first join-node (in a postorder traversal) in the current state is marked as *current node,*
- The current node is repeatedly swapped with its ancestors, until the root of the JT is reached, maintaining the legal swaps performed in this process. The least expensive state produced by a legal swap replaces the current state; the SCAN() restarts at the first join-node of the new current state. If none of the swaps is legal, the procedure marks the next join-node in the postorder traversal as the current node and restarts. The procedure ends when there are no more join-nodes to visit.

The hybrid technique uses SCAN() as a move, in the sense that SCAN() constructs a state with lower cost than the one on which SCAN() was applied. On the other hand, the *swap()* operations initiated by SCAN( ) are tested for legality, and thus observed as submoves, too.

The SCAN( ) move is further enhanced by ignoring improvements less than a certain percentage over the cost of the current state, so that the overhead of restarting the SCAN() in order to obtain a minor improvement is avoided. Experimentation showed that the value of 0.99 can be used as a cost improvement threshold without affecting the optimality of the solution.

The swap() is applied exhaustively on the nodes of the JT corresponding to a start state. This ensures that the neighbourhood of the start state, as determined by the *swap()* move, is scanned thoroughly for execution plans. Legal swaps are used to select the one with the lowest cost, and illegal ones are ignored, so that the execution plan produced for a start state is a “local optimal plan". It is local because it is optimal within the specific neighbourhood. This plan roughly corresponds to a local minimum produced by a method like iterative improvement.

The swap( ) is restricted to the neighbourhood of a start state. which forms a small part of the search space. Therefore, it is combined with the *reroot()* move. *reroot* defines a neighbourhood of N very different states: this neighbourhood could be classified as "sparse". For each state in this sparse neighbourhood, the *swap()* move produces states relatively close to each other i having differences on some locations of the corresponding trees and on the selected algorithms): those states build a neighbourhood that can be classified as "dense". Those dense neighbourhoods are exhaustively scanned.

For small trees, this technique is exhaustive, since the sparse neighbourhood defined by *reroot()* combined with the dense neighbourhoods defined by *swap()* for the start states do cover the whole search space. Then, the behaviour of the proposed technique is close to that of an exhaustive technique. For larger trees. the technique departs gradually from the exhaustive approach, since the union of the sparse neighbourhood and the dense ones leaves areas of the search space unexplored. Then the behaviour of the proposed technique comes closer to the behaviour *of* iterative improvement.

A language diagram of the hybrid technique is shown in Table I.

Tab. I. The algorithm of the hybrid technique

```
MAIN()
current_optimal := compute_hybrid_tecnnique_minimum(start_state = initial_state)
FOR new_root IN joins_in_tree DO
    new_tree := reroot(tree = initial_state, root = new_root);
    candidate_optimal := compute_hybrid_technique_minimum(start_state = new_tree);
    IF cost(tree = current_optimal) > cost(tree = candidate_optimaI) THEN
        current_optimal := candidate_optimal;
    ENDIF
ENDFOR
optimum := current_optimal
END MAIN:

PROCEDURE compute_hybrid_technique_minimum(start_state)
current_state := start_state:
target_node := first postorder_join(tree = current_state);
REPEAT
    done := TRUE;
    WHILE NOT root_reacned(tree = current_state, node = target_node);
        test_state := swap(tree = start_state. node =target_node)
        IF cost(tree = current_state) > cost(tree = test_state) THEN
            current_state := test_state;
            done := FALSE;
        ENDIF
        IF done THEN /* No changes made */
            target_node := next_postorder_node(tree = current_state, node = target_node);
            IF target_node <> NIL THEN
                done := FALSE; /* Continue with next node */
            ENDIF
        ELSE /* At least one change has been made */
            target_node := first_postorder_node(current_state);
        ENDIF
    END WHILE UNTIL
done:
RETURN current state
END_PROCEDURE.
```

4.3 Outline of the iterative-improvement technique

The alternative method used in our model is based on conventional iterative improvement. described thoroughly in [9. 12]. The rationale behind the development of this local search method is the optimisation of join queries for which the overhead of the almost exhaustive technique proposed in this study is inacceptable. For the purposes of this study, we briefly outline this second technique, in order to perform comparisons on the behaviour of both techniques, and to draw conclusions on the query size threshold, above which techniques of exhaustive nature are inappropriate.

Similarly to the hybrid technique, our version of a local search method uses the *reroot()* operation for the construction of the start states. However, instead of applying the exhaustive SCAN() move on each start state, the iterative-improvement technique employs the ROLL() move, which is comprised by a series of *swap()* operations. Each execution of ROLL() is checked for legality and is accepted only if legal, i.e. if it produces a state of less cost than the one it was applied on.

The ROLL() move repeatedly swaps a randomly selected node with a number of nodes adjacent to it. The number of swaps, as well as the direction of the ROLL() (towards the root or towards the leaves, left or right) are also selected at random. The legality of each individual swap() is not checked.

ROLL( ) is applied on the same sparse neighbourhood defined by *reroot( ),* as used in the hybrid technique, but the neighbourhood it defines per start state is sparser than the one defined by SCAN(). ROLL( ) is initiated for a start state, and produces a new state in each initiation, to be used for the next ROLL( ). Upon completion of the sequence of ROLL() moves, a local minimum has been constructed for the start state.

The signal for the completion of a sequence of ROLL() moves must be twofold: Actually. the moves must stop when a local minimum is produced. since no move applied on the local minimum will construct a state of a lower cost. However, an upper limit must be set to the length of the sequence of moves starting at a start state, since it may be the case that no locale minimum can be constructed for a neighboorhood within an acceptable optimisation overhead.

Therefore. the iterative-improvement technique only performs for a time span of: $T_{seq} = c * n^3$ where *n* is the number of joins on the join tree and *c* is a constant corresponding to the time required for the optimisation of a threshold number of joins using the sequential version of the hybrid technique [10]. Below this threshold, the overhead of an exhaustive technique is still acceptable: for the threshold number both techniques must have the same overhead.

Upon expiration of the time limit, any interrupted ROLL( ) sequence declares as local minimum the state with the lowermost cost produced insofar, whereas the state with the minimum cost among the constructed local minima is declared as the global minimum. The time limit is analogous to that proposed by Swami and Gupta [13] for the construction of a single local minimum ($T_{single} = a * n^2$) for each of the *n* starts states.

4.4 Usage of parallelism for the hybrid technique

Both optimisation techniques described insofar are clearly parallelisable. The start states produced by *reroot()* can be processed independently by the SCAN(), and or the ROLL() move. Moreover, since the arguments of *reroot()* are the single initial state and one JOIN-node, the construction of the different start states can also be performed in parallel. Therefore, we organise the optimisation scheme as follows:

Phase 1. One master process, which acts as "coordinator", obtains the JT corresponding to the initial state. This JT is normally the output of a previously invoked query processing module, like the parser/pre-optimiser we proposed in [11]. The initial state is then produced by assigning execution algorithms *to* the tree nodes.

Phase 2. The coordinator activates n processes. where n is the number of JOIN-nodes in the join tree, and forwards to each of those processes the initial state, its cost and one node N, which will be used as argument to *reroot( ).* The coordinator assures that each of the n processes receives a different node N, and that one of them will maintain the initial state as is, since the initial state is also a start state.

Phase 3. Each of the n processes invokes *reroot( )* and obtains a start state by assigning algorithms to the nodes of the produced JT. Thereafter, SCAN( ), or ROLL( ), is activated. which computes a local optimal plan (LOP) for the start state. The cost of this LOP, denoted as TLOP, is returned to the coordinator.

Phase 4. The coordinator gathers all T_LOPs of the local optimal plans computed in parallel, selects the minimum among them and then obtains the state corresponding to that T_LOP from the process that has computed it. This state is declared as -the optimal execution plan for the query.

The coordinator process can be optionally split in two processes, one "initialiser", which performs the tasks of Phase i and activates *n-1* processes at the beginning of Phase 2, and one "selector", which gathers the cost estimations at the end of Phase 3 and obtains the optimal plan in -Phase 4. This approach is more suitable for parallel environments where bidirectional communication among processes is not very easy to establish or is too expensive to maintain, in the following, we observe the coordinator as one logical process.

The aforementioned parallel model requires the activation of *n* processes. Since the coordinator process is idling during Phase 3, which is the potentially longer one, and since one of the *n* processes can avoid rerooting because it operates on the initial state, it is more convenient to have the coordinator invoke *n-1* processes and participate in Phase 3 itself, using the initial state as its start state. Thus, a total of *n* processes is required.

In the parallel version of the iterative-improvement technique, the local minima and the global minimum are treated like the Local Optimal Plans and the Global Optimal Plan of the hybrid technique respectively: local minima are constructed in parallel during Phase 3 and forwarded to the coordinator with gradual filtering during Phase 4, until the global minimum is selected among those reaching the coordinator.

4.5 **Overhead** of parallel optimisation

The parallelisation of the optimiser dramatically decreases the optimisation overhead for the Hybrid Technique to the cost of scanning the dense neighbourhood which produces the most slowly constructed LOP. Although the dense neighbourhoods can be very dissimilar and the cost of scanning each one may vary, the total optimisation cost is drastically reduced.

However, the parallelisation of the optimisation technique adds an overhead that does not appear in the sequential version: the transfer of the initial state towards the *n-1* processes during Phase 2, the transfer of the T_LOPs towards the coordinator during Phase 3 and the final transfer of the optimal plan towards the coordinator in Phase 4 add communication overhead. Moreover, the assignment of algorithms to the tree nodes for the construction of a state, as well as the computation of a state's cost, require meta-information on the relations, which is traditionally maintained in the data dictionary. The overhead of *n* processes accessing the disc in the parallel version versus a single process retrieving the dictionary in the sequential version is considerable. In order to reduce the additional overheads, the algorithm of the parallel technique is further enhanced.

I/O overhead. The meta-information of the data dictionary which is required by the optimisation technique is normally small enough to fit in main memory. Since the optimisation technique is a CPU-intensive application, the available memory of a processor can be reduced to have the (necessary part of the) dictionary reside in memory without affecting the performance of the technique. Thus, the I/O overhead is reduced to the initial loading of the dictionary file.

The I/O overhead for loading the data dictionary to the *n* processes is reduced by having the data dictionary read from disc once by the coordinator and be subsequently broadcasted to the *n-1* processes. Thus, the I/O overhead of the parallel technique becomes equal to that of the sequential version.

Communication overhead. The parallelism that can be achieved by the parallel model is obviously limited by the available processing power. If *n* is larger than the number of available processors *p,* then multitasking must be performed. In its simplest form, multitasking would imply the invocation of *n* processes and their even allocation on the *p* available processors. However, this would result in further communication overhead, by having the *n/p* processes residing on one processor compete for a route to communicate with the coordinator.

Since *n* and *p* are known, when the parallel optimisation technique is initialised, a better alternative to simple multitasking is feasible: the coordinator can activate *p* processes and specify that each of them constructs *n/p* start states and applies Phase 3 sequentially on each of them. The CPU cost is identical to that yielded by simple multitasking. However, communication overhead is reduced since the coordinator has to interact with *p-1* instead of *n-1* processes across the same routes.

According to the aforementioned enhanced scheme of processes allocation to processors, the interaction among the processes changes as follows:

During Phase 2, the coordinator forwards to each processor the node_ids of all nodes that should become roots of the start states to be constructed by each processor. If the number of available processors *p* is greater or equal to *n,* then *n* processors are used, each one retrieving a single node_id; if $p < n$, then each processor receives approximately *n/p* node_ids. So, the technique employs $min\{n, p\}$ processors, namely the coordinator + $p_{additional}$ processors, where $p_{additional} = min\{n, p\} – 1$.

During Phase 3, each processor constructs a local optimal plan for each start state it has created. and selects the least expensive among them. The other ones are discarded. So, at the end of Phase 3 exactly $\mathrm{p}_{\mathrm{additional}}$ T_LOPs are returned to the coordinator.

The communication overhead of Phase 2 is reduced by having the coordinator broadcast the initial state and the dictionary, instead of communicating with each of the $\mathrm{p}_{\mathrm{additional}}$ processors separately. The node_ids (small integers) of the new roots need not be broadcasted: they can he passed as parameters to the tasks implementing the processes. Since the broadcast facility uses the $\mathrm{p}_{\mathrm{additional}}$ routes simultaneously, the communication overhead of Phase 2 is the cost of using the most expensive among those routes.

Assuming that all routes connecting any two processors are composed of links (channels) having the same cost and bandwidth. the most expensive route is the longest one. Let t_{net} be the time required for the information unit to be transferred across a link. Let *Li* be the number of links across the i-th route. Since pipeline is possible, one link continuously propagates transfer units to the next one in a pipeline mode. Then the communication overhead of Phase 2 is:

$$\mathrm{T}_{\mathrm{Phase2}} = \left(\max_{\mathrm{i}} \mathrm{Li}\right) * \mathrm{t}_{\mathrm{net}} + (\,\mathrm{size\_of\_initial\_state} + \mathrm{size\_of\_dictionary}\,) * \mathrm{t}_{\mathrm{net}}$$

where the factor $(t_{net} * max\ L_i)$ is the propagation time for initialising the longest route (pipeline startup time).

Since all states have the same number of nodes and the same size of contents, all states have the same size, denoted as S_{n}. network transfer units, which depends on the number *n* of nodes of the JT. Moreover, the size of the dictionary is fixed for a database and equals to DD transfer units. Thus:

$$T_{Phase2} = \left(\max_{i} Li\right) * t_{net} + (S_n + DD) * t_{net}$$

The communication overhead at the end of Phase 3 is the cost of using the aforementioned routes in the opposite direction. Since the routes are used by $p_{additional}$ senders towards the same receiver, some processors would compete for (parts of) an *a* route, unless the parallel system can place the $p_{additional}$, processors around the coordinator in a star topology. The competition is avoided, though, by the following enhancement:

Let

Pi — Pi-1 — ... — P1 — Coordinator

be a route connecting the processors i, i-1, ..., 1 to the coordinator. Furthermore, let T_LOP be the cost of the less expensive Local Optimal Plan LOP, constructed by processor P. Normally' T_LOP, should pass from processors P_{i-1}, ..., P_1 before reaching the coordinator. However, processor P_{i-1} , could compare T_LOP_i with its own T_LOP, and forward the smallest of the two. If this filtering is performed by each processor across a route towards the coordinator, then only one T_LOP passes across any link towards the coordinator, which obtains the minimum among them. The overhead of one additional comparison per processor per route is completely negligible. Then. the communication overhead of Phase 3 is:

$$T_{Phase2} = \left(\max_{i} Li\right) * \text{double\_precision\_size} * t_{net}$$

Note that a cost value T_LOP is not propagated across the route in a pipeline mode, since it is stopped by *each* processor P across the route it follows towards the coordinator and is compared to the T_ LOP. If processor P has not finished constructing its own LOP, then T_LOP, LOP are kept in the memory of processor P_{i-1}, until they can he submitted to processor P_i. Since processor P_i has already finished processing, its memory can he used to store the data. This delay is part of the CPU overhead caused by the optimisation process itself.

So, the communication overhead of Phase 4 is the transfer of the optimal plan from the processor that constructed it towards the coordinator. In order to avoid an additional message from the coordinator asking a specific processor P_x to submit the local optimal plan, all LOPS of the processors are forwarded across the routes towards the coordinator, in the same way as the T_LOPs of Phase 3:

- P forwards LOP towards the coordinator after having forwarded T_LOP.
- Processor P_{i-1} compares T_LOP to T_LOP_{i-1} in Phase 3: if the former is smaller than the latter, then P_{i-1} forwards LOP_i towards the coordinator in Phase 4, discarding its own LOP_{i-1}. Otherwise, LOP_i is ignored when it reaches processor P_{i-1}, which forwards LOP_{i-1} towards the coordinator. So, only one LOP is transferred across any link of any route towards the coordinator. The LOP that reaches the coordinator is the smallest one, which becomes the optimal plan. Since the optimal plan is just another state of the join tree, having size = S_n transfer units, the communication overhead of this phase is:

$$T_{Phase4} = \left(\max_{i} Li\right) * S_n * t_{net}$$

The iterative-improvement technique shows similar communication and I/O overhead to the hybrid technique, except that Phase 4 is signalled simultaneously for all processors by the expiration of the time span T_{par}. Thus, the pipeline of Phase 4 is not broken by the CPU delays incurring in the hybrid technique. This fact contributes to the difference in the CPU overhead of the two techniques, while the relative communication overheads are not affected.

5 IMPLEMENTATION OF THE PARALLEL OPTIMISER ON A TRANSPUTER MACHINE

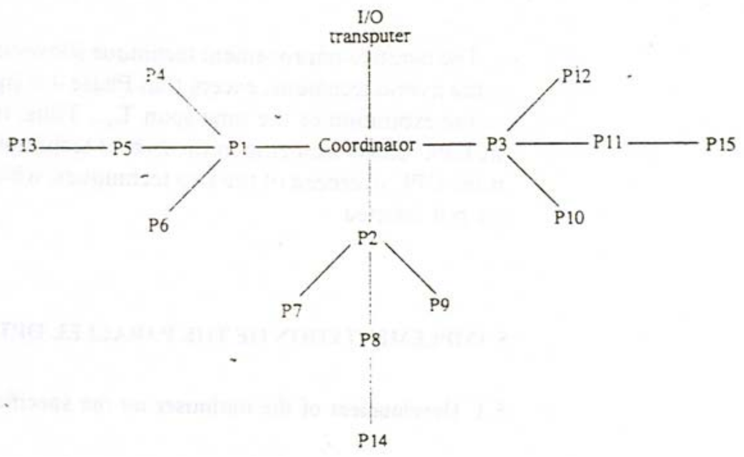
5.1 Development of the optimiser on the specific architecture

The proposed parallel optimisation technique for queries with many joins is currently implemented on a 20-transputer machine operating as a back-end to a SUN server, which offers the secondary storage. The transputers have private memories of 4 Mbytes each and communicate across a network offering a bandwidth of approximately 4 Mbits sec.

Each transputer can be connected to other transputers using 4 software-reconfigurable links. Communication with the host machine is ensured by 4 transputers hardwired to the host: those transputers are mainly used to access the secondary storage and are therefore called "I/O transputers".

The hybrid technique is being developed under the Helios3[3] operating system, using C for the development of the code to run on a single processor, and CDL4[4] for the topological description of the tasks that run in parallel. In this environment, we are building two optimisation techniques, namely the hybrid technique for join queries with tens of join operators proposed in this study, and the iterative-improvement technique, designed for queries containing a very large number of joins, as well as set operators and aggregate functions.

For the hybrid technique, we configure the transputers into a topology corresponding to a star of stars (Fig. 1), having the coordinator linked to one of the I/O transputers. The topology of the tasks comprising the optimisation technique is described in a CDL script, which defines the coordinator, the $p_{additional}$ tasks and the interaction among them. Since the desired communication among those tasks is best achieved in a star topology,the CDL script used by the hybrid technique maps one-to-one the topology of the tasks to the star configuration. Actually, this script specifies the transputer on which each task will run, placing the coordinator on the transputer denoted as "COORDINATOR" in Fig. 1, so that communication with the I/O transputer is established using the smallest possible route.



3 Helios is a trademark of Perihelion Software Limited.
4 The CDL language was desiged by Andy England and Charlie Grimsdale.

Fig. 1. Network configured into a star-of-stars composed by one I/O transputer and 16 transputers denoted as Coordinator and P1. P2. .... P15

In the general case of a transputer network with *p* transputers (beyond the I/O transputer), if a join tree requires the activation of less than *p* tasks (including the coordinator), then only the necessary transputers are used, preserving the star-of-stars topology. Otherwise, all *p* transputers are activated, each one computing *n/p* local optimal plans, where *n is* the number of joins in the JT.

The modules of the parallel tasks activated by the technique's script are written in the C language. The following modules are loaded within each task:

1. The *reroot()* and *swap()* operations for the construction of local optimal plans.
2. Computation of the cost T_LOP for a local optimal plan LOP.
3. Computation of the minimum among a number of T_LOPs, that are either cost values of LOPs constructed by the task, or cost values received from other tasks across the network.
4. Reception of a T_LOP from the network.
5. Reception of a LOP from the network.
6. Submission of a T_LOP to the network towards the coordinator.
7. Submission of a LOP to the network towards the coordinator

The coordinator task has two additional modules:

C1. Reception of the initial state corresponding to the initial query tree having algorithms assigned to its nodes. The pre-optimiser we proposed in [11] is used *for* the transformation of a query into an equivalent query tree.

C2. Broadcasting of the initial state and the data dictionary to the other tasks.

The node_ids of the nodes that should be used as neew roots in *reroot()* to produce the start trees of each task are passed as parameters to the C code of the task module I.

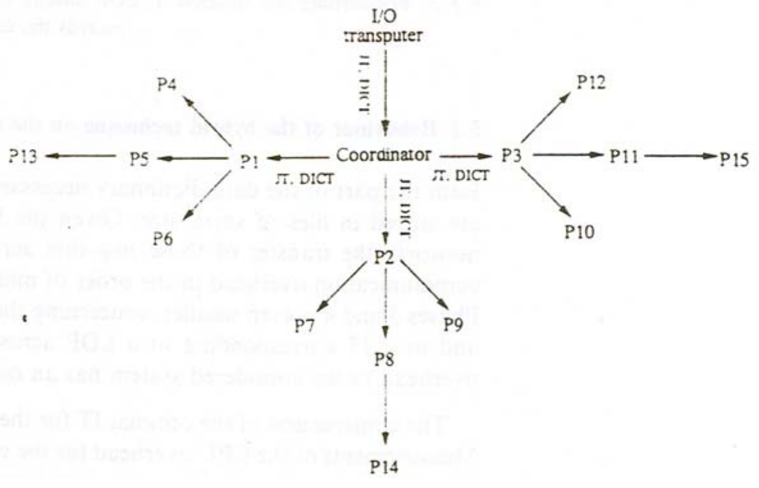


Fig. 2. Broadcasting of the data dictionary DICT and the start state JT for a join tree requiring transputers

The broadcasting of the data dictionary and the initial state during Phase 2 is depicted in Fig. 2. The state and the dictionary are received by the coordinator and forwarded to the transputers of the central star. Further forwarding takes place in the same way.

The cost of the least expensive local optimal plan computed by each task is submitted across a route of the star-of-stars (Fig. 3), where T_LOP(i) is the minimum among the

T_LOPs computed by the tasks on P_1 while T_LOP(i_1, i_2, ...,i_k) is the minimum among T_LOP(i_1) computed on P_1 and the T_LOP(i_2), ..., T_LOP(i_k) received across the network. The submission of a LOP having cost T_LOP takes place in a similar way.

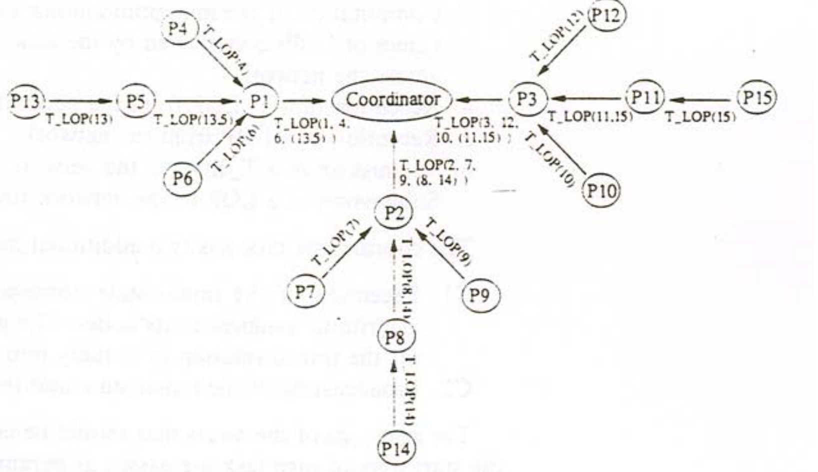


Fig. 3. Forwarding the smallest T_LOP among those computer by the tasks across each route towards the coordinator

5.2 Behaviour of the hybrid technique on the specific architecture

Both the part of the data dictionary necessary for the optimisation and the initial state are stored in tiles of small size. Given the high bandwidth of the parallel machine's network, the transfer of those two files across the network during Phase 2 shows a communication overhead in the order of milliseconds. The communication overhead of Phases 3 and 4 is even smaller, concerning the submission of a double precision number and of a JT corresponding to a LOP across the network. Thus, the communication overhead in the considered system has an order of magnitude O($10^{-3)}$ seconds.

The construction of the optimal JT for the join query is a CPU-intensive application. Measurements of the CPU overhead for the computation of the optimal plan for queries with N joins have revealed an exponential increase. Since the CPU overhead ranges over seconds (and tens of seconds), the communication overhead is orders of magnitude lower than CPU cost and can be safely ignored. Thus, the parallelisation of the optimisation technique reduces the optimisation overhead to the CPU cost of computing the most slowly constructed LOP. Although this cost also increases exponentially, the border beyond which the technique's overhead is 'unacceptable is shifted towards larger values.

As shown in Fig. 4. the CPU overhead for the sequential hybrid technique quickly increases to many tens of seconds for queries with more than 10 joins, while the time required to construct the most slowly constructed LOP from a start state increases at a slower pace with the loin tree size. It must be noted that in order to be closer to the query types usually issued towards an RDBMS, the experiment queries contain joins, selections and projections PSJ-queries; however, the performance of the proposed technique is affected by the number of joins only.

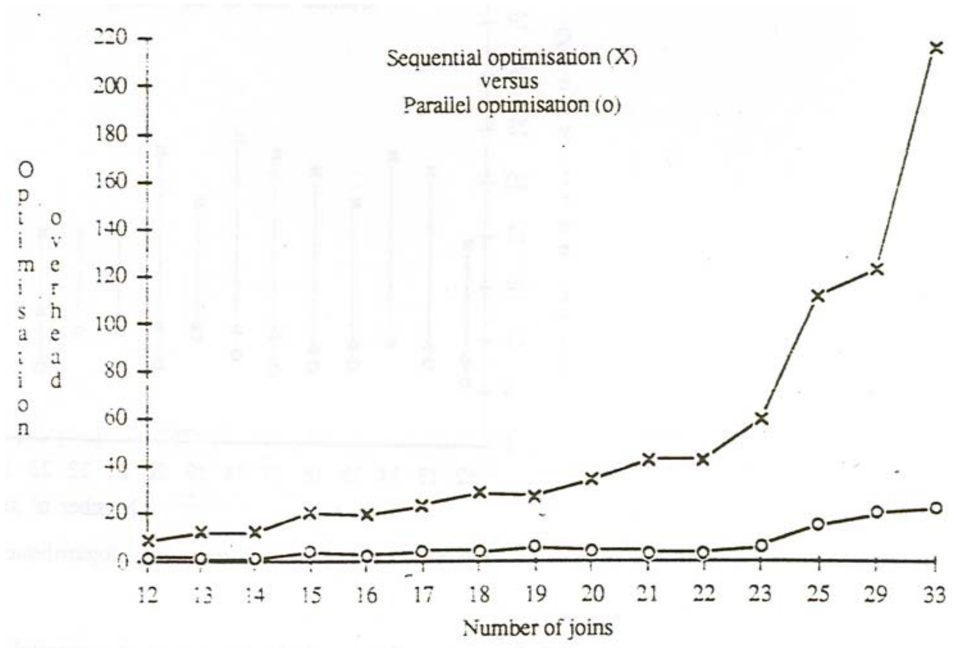


Fig. 4. The time overhead (in seconds) of the sequential and the parallel versions of the hybrid technique for a set of join queries

We show the cost improvement achieved by the hybrid technique for the set of queries of Fig. 4: the initial and optimal cost per query is computed using the cost model we propose in [10], which produces a normalised amount analog to the expected execution time. The same figure contains the optimal cost achieved by the iterative-improvement technique. As expected, the optimality achieved is lower than that ensured by the almost exhaustive search of the hybrid technique.

For these comparisons, T_{par} was set equal to *n/2* by using a constant *c* = 15 30 corresponding to an overhead of 30 seconds for the initial threshold of a 15-join query using the sequential hybrid technique. Since T_{par} must he adequately high to allow the construction of at least one local minimum per processor, the iterative-improvement technique is not appropriate for very small queries. The parallelism further shifts upwards the threshold below which the hybrid technique shows still an acceptable overhead. As indicated in Fig. 4, the threshold above which the hybrid technique must be replaced by the iterative-improvement technique shifts towards queries with as many as 30 joins, beyond which the curve *of* CPU overhead for the hybrid technique shows clearly the trend for exponential increase[5].

[5] The optimiser using the proposed technique is also addressing the same category of databases [9]

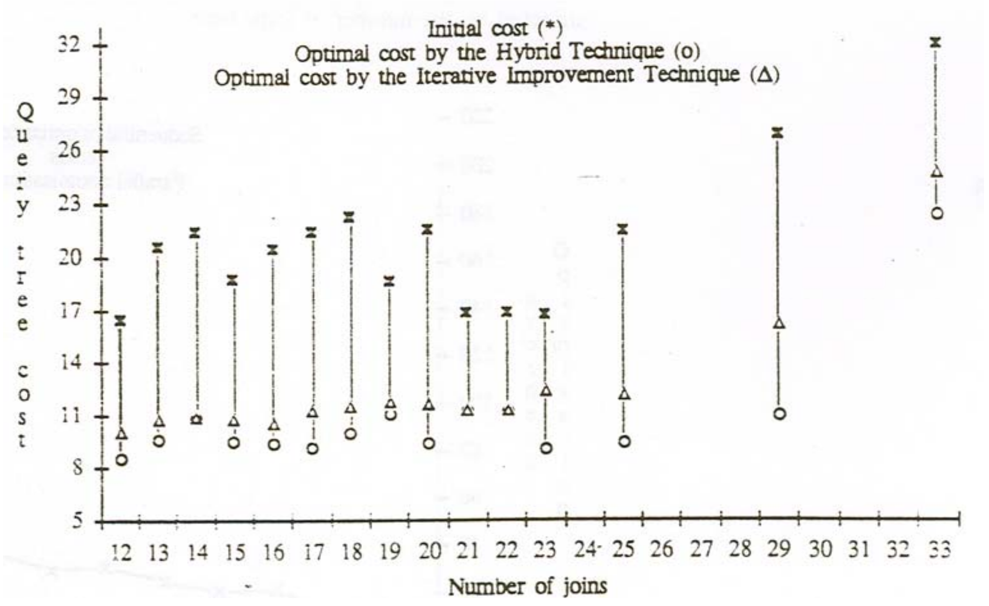


Fig. 5. Initial and optimal query cost (logarithmic scale) for a set of join queries

5.3 Execution of the optimised query on the parallel machine

The output of the optimisation phase is the optimal query execution plan, which is also executed in parallel. Parallel query optimisation and parallel query execution are performed on the same machine, since one of the fundamental aims of our work has been the utilisation of the processing power, as offered for query execution, during the preceding query optimisation phase. This concerns the fundamental characteristics of the processors, i.e. a MIMD machine with private processor memories (and with shared or private disc peripherals: this does not affect much the CPU-bound optimisation process). On certain machines, the network configuration used for query optimisation is also appropriate for query execution.

The optimal query execution plan produced by the optimiser is mapped into a task graph, subsequently loaded into the transputer network and executed to produce the

The constant c used by the iterative-improvement technique does not need to he readjusted after the shifting of the threshold query results. This task graph is actually a directed tree, in which each node corresponds to a unary or binary operation of relational algebra, i.e. selection, projection, join, union, intersection, difference. The leaf nodes of this tree access the relations residing on disc, while intermediate tree nodes retrieve data streams in pipeline mode. The root node of the tree corresponds to the task which presents the query results to the user. According to this tree structure, each node in the task graph communicates with at most three other tasks, i.e. one or two child-task which supply the input relation(s), and one parent task which receives the output relation.

On the specific parallel machine, the transputer topology defined for the query optimization phase is suitable for the query execution phase, so that no reconfiguration of the network is necessary between query optimisation and query execution. For the execution phase, the root node of the tree is placed on the I/O transputer and all other nodes are placed in such a way that they are adjacent to the nodes, which they receive input from or forward results to. Since all transputers have four links and up to three of them are used for task communication, some links remain free and are used as "shortcuts" for forwarding the disc relations to the leaf nodes. The forwarding of the disc relations to the requesting nodes is done "on demand", i.e. the relation pages are forwarded to the appropriate task when they are requested for and not "in advance", since this would cause unnecessary network traffic due to forwarding pages which will not be used.

The configuration of the transputer machine for query optimisation may form a bottleneck during query execution of task trees with many leaves, corresponding to many relations to be retrieved from disc. In that case, a reconfiguration of the network would be preferable prior to query execution, in order to place as many leaf nodes as possible to I/O transputer(s) and to reserve all available I/O transputers of the whole network. However, the applicability of a parallel machine for our model is not affected by selecting different software-defined topologies for query optimisation and query execution.

6 CONCLUSIONS

In this study, we have presented a hybrid technique of almost exhaustive nature for the optimisation of queries with tens of join operations. The technique uses characteristics of both exhaustive strategies and of strategies based on iterative improvement: it identifies a sparse neighbourhood of the initial state within the solution space and builds a dense neighbourhood (scanned exhaustively) for each state of the sparse neighbourhood. For small join trees, the combination of the sparse neighbourhood and the dense ones corresponds to the whole solution space: for larger trees the technique deviates smoothly from the exhaustive approach.

The parallelisation of this optimisation technique reduces the problem of finding the optimal plan for a join tree to that of computing a local optimal plan for one start state of the join tree. The available processing power is used to compute one local optimal plan for each start state of the sparse neighbourhood in parallel. The parallel version of the hybrid technique is primarily designed for databases operating on low parallelism architectures composed of processors with private memories and either shared or private secondary storage, which are linked together across a high bandwidth network. For parallel machines and for LANs with high bandwidth channels, the communication overhead is reduced by various enhancements and becomes negligible. So, the overhead of the optimisation technique is reduced proportionally to the number of processors employed by the technique.

The usage of parallelism during the optimisation phase both reduces the optimisation overhead into acceptable boundaries and shifts upwards the query size limit, beyond which techniques based on randomised algorithms should be utilised. Those techniques can also be improved by using the offered parallelism. Therefore, one such technique, the iterative-improvement technique, was compared to the hybrid technique in our model, stressing the relative behaviour of the two techniques in terms of time overhead and optimality of results.

Parallel optimisation for parallel query execution is a very promising approach to the recently emerging problem of optimisation of very large queries. We are currently working on the parallelisation of both the hybrid and the iterative-improvement techniques for queries containing not only a large number of joins, but also set operators and aggregates, and we are studying the behaviour of the techniques on the experimental site described in Section 5.1. Since this site corresponds to a back-end machine with shared secondary storage, one of the future extensions of the model is the consideration of the impact of non-shared discs to the optimisation techniques.

REFERENCES

[1] DEWITT. D. J. - Gray.J: Parallel database systems: The future of database processing or a passing fad? ACM-SIGMOD. 1990. Vol. 19. No. 4. pp. 104-112.

[2] IOANNIDIS.E. —Wong, E.: Query optimization by simulated annealing. Proc. of the ACM-SIGMOD. International Confereee on Manag^{e}ment of Data. San Francisco 1987. pp. 9-22.

[3] Ioannidis. Y. E. Kang:. Y. C.: Randomized algorithms for optimizing large join queries. Proc. of the ACM-SIGMOD International Conference in Management of Data. Atlantic City 1990. pp. 312 - 321

[4] Ioannidis Y.E., Kang Y.C. : Left-deep vs. bushy trees: An analysis of strategy spaces and implications on query optimization. Proc. of the ACM-SIGMOD International Conference on Management of Data. Denier 1991. pp. 168-177.

[5] KIM. W.: On optimizing an SQL-like ested query. ACM-TODS. 1982. Vol. 7. No. 3. pp. 443-469.

[6] Mikkilineni. K, Su S.: An evaluation of relational join algorithms in a pipelined Query processing environment. IEEE Trans. of Software Eng. 1988. Vol. 14. No. 6. pp. 838-848.

[7] SELINGER. P. G. et al: Access path selection in a relational database management system. Proc. of ACM-SIGMOD International Conference on Management of Data. Boston 1979. pp. 23-34.

[8] SHAPIRO. L. D.: Join processing in database systems with large main memories. ACM- TODS. 1986. Vol. I1. No. 3. pp. 239-264.

[9] Spiliopoulou, M., HATZOPOULOS. M.: Parallel optimization of large join queries with set operators and aggregates in a parallel environment supporting pipeline. Submitted for publication.

[10] Spiliopoulou, M. — Hatzopoulos M.—Vassilakis C.: Cost and behaviour of nested and canonical SQL-queries in a parallel environment supporting pipeline. Submitted for publication.

[11] Spiliopoulou, M.-HAtzopoulos. M.: Translation of SQL-queries into a graph structure: Query transformation and pre-optimization issues in a pipeline multiprocessor environment. Information Systems 1992. Vol. 17 No.

[12] Spiliopoulou, M.: Parallel optimisation and execution of queries towards and RDBMS in an environment of parallel and pipelined processing. Ph.D. thesis. Department of Informatics. University of Athens. March I992 (*in* Greek).

13] Swami. A. - GUPTA. A.: Optimization of large join queries. Proc. of the .ACM-SIGMOD International Conference on Management of Data. Chicago 1988. pp. 8-17.

141 SWAMI. A.: Optimization of large join queries: Combining heuristics and combinatorial techniques. Proc. of the ACM-SIGMOD International Conference on Management of Data, Portland I989. pp. 367-376.

Manuscript received September 1992: revised December 22. 1992

Maria Spiliopoulou was born in Athens. Greece. She received the B.S. degree in mathematics from the University of Athens. 1986. and the Ph.D. degree in informatics from the University of Athens. 1992. She has participated in national and multinational (E.E.0 ESPRIT and DELTA) projects with the University of Athens. Her current research interests include parallel query optimisation in relational and object-oriented databases, object-oriented query languages, and modelling of multimedia and hypermedia applications on object-oriented databases. Dr. Spiliopoulou is a member of the Association for Computing Machinery

Michael HATZOPOULOS was born in Athens. Greece. He received the 3.S. degree in mathematics from the University of Athens, in 1971, and the M.Sc. and Ph.D. degrees in computer science from the University of Loughborough. England in 1972 and 1974, respectively. He has worked as Research Associate of the Department of Mechanical Engineering. Loughborough University, as lecturer in the University of Athens and as Visiting Associate Professor of Michigan Technological 1 University. Houston. USA. In 1989. he joined the Department of Informatics University of Athens as Associate Professor. and he is professor of the department since 199I. His current research interests include physical database design, distributed processing, multimedia information systems and design of parallel algorithms. Dr. Hatzopoulos is a member of the Association for Computing Machinery and the IEEE Computer Society.

Costas VASSILAKIS was born in Arta. Greece. He received the B.S. degree in informatics from the University of Athens in I990 and is now a Ph.D. Candidate. He has participated in national and multinational (E.E.C. ESPRIT and DELTA) projects with the University of Athens. His current research interests include distributed systems, artificial intelligence, simulation and object-oriented programming.