On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces *

Rosana S.G. Lanzelotte[2,1], Patrick Valduriez[1], Mohamed Zaït[1]

[1]Projet Rodin, INRIA, Rocquencourt, France
[2]Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Brazil
e-mail: firstname.lastname@inria.fr

Abstract

The cost of query optimization is affected by both the search space and the search strategy of the optimizer. In a parallel execution environment, the search space tends to be much larger than in the centralized case. This is due to the high number of execution alternatives which implies a significant increase in the optimization cost. In this paper, we investigate the trade-off between optimization cost and parallel execution cost using the DBS3 parallel query optimizer. We describe its cost model which captures all essential aspects of parallel executions. We show how the cost metrics imply a significant increase in the search space and optimization cost. However, instead of restricting the search space, which may lead to loosing better plans, we reduce the optimization cost by controlling the search strategy. We extend randomized strategies to adapt well to parallel query optimization. In particular, we propose Toured Simulated Annealing which provides a better trade-off between optimization cost and quality of the parallel execution plan.

1 Introduction

Query optimization refers to the process of producing an *optimal* execution plan for a given input query, where optimality is with respect to a cost function to be minimized. An "optimal" plan has the least cost among all equivalent plans. The investigation of equivalent plans is driven by the optimizer search strategy within a "search space" in which each point is a possible plan. Execution plans are typically abstracted in terms of processing trees (PTs) to capture in a compact way the aspects that are essential for cost estimation and optimization. PTs express executions which involve *inter-operation* or *intra-operation* parallelism.

Parallel optimization is made difficult by the necessary trade-off between optimization cost and quality of the generated plans (the latter translates into query execution cost). The optimization cost is affected by both the size of the search space (i.e., the number of possible PTs) and the search strategy, which can be more or less exhaustive. Compared to centralized query processing [Se79], the major problem to be addressed is dealing with a large space of parallel execution plans. In the centralized case, the search space gets obviously larger as the query is more complex, e.g., includes many joins [Sw89], deals with complex objects or includes recursion [LVZ92]. Even for a reasonable query (e.g., with less than 10 joins), a parallel execution model yields a large variety of execution plans due to the various sources of parallelism. Thus, optimization of reasonable queries may be intractable if an exhaustive search strategy is used in conjunction with a large space of parallel executions.

To reduce optimization cost, most centralized [Se79] and parallel [SD90, HS91] optimizers restrict the search space to linear PTs. In [ZZB93], we showed that this may lead to loosing the optimal PT. Also, linear PTs cannot capture independent parallelism, where operations involving disjunct sets of operands are executed in parallel. An alternative approach is to use a non-exhaustive search strategy. With this objective, randomized search strategies have been proposed to improve a start solution until obtaining local optima. Examples of such strategies are Simulated Annealing (SA) [IC91] and Iterative Improvement (II) [Sw89]. Randomized strategies do not guarantee that the best solution is obtained, but avoid the high cost of optimization.

In this paper, we investigate the trade-off between optimization and execution costs in a parallel database system. Our framework is the DBS3 parallel database system [BCV91], which is developed by Bull and INRIA

*This work was partially funded by the Esprit project IDEA, and the PRC BD3 of the French Ministry of Research.

Proceedings of the 19th VLDB Conference
Dublin, Ireland 1993

as part of the EDS and IDEA Esprit projects. DBS3 implements a distributed memory (shared-nothing) execution model on a shared-memory multiprocessor. However the DBS3 optimizer has a parameterized cost model which can be set to either shared-memory or shared-nothing [ZZB93]. In the later case, the execution plan can be executed on the EDS machine [EDS90]. In this paper, we use the shared-nothing cost model since it is the most general. The DBS3 optimizer explores both sources of parallelism, inter-operation and intra-operation. It relies on a cost metrics that captures all the relevant parallel aspects, e.g., relation fragmentation and scheduling. When compared through this metrics, many more tentative PTs are kept during the search, thereby increasing significantly the optimization cost.

The DBS3 optimizer uses efficient non-exhaustive search strategies [LV91] to reduce query optimization cost. Using a realistic application, we measure the impact of parallelism on the optimization cost and the optimization/execution cost trade-off using several combinations of search space and search strategy. The major contribution of this paper is an extension of SA called Toured Simulated Annealing (TSA), to better deal with parallel query optimization. In all experiments, TSA yields the best optimization/execution cost ratio. The conclusion of our experiments is that controlling the search strategy to reduce the optimization cost is usually better than restricting the search space.

The rest of the paper is organized as follows. Section 2 describes the different search spaces and related cost model dealt with by the DBS3 optimizer. Section 3 presents the optimizer search strategies and proposes several variants to enhance their effectiveness. Section 4 shows the experiments and measurements. Section 5 concludes.

2 Search Space and Cost Model

The main components of a query optimizer are the *search space*, the *cost model* and the *search strategy*. Our approach enforces the independency between these three components, which enables adapting the optimizer to different requirements. In this section, we describe the context of the DBS3's parallel query optimizer [ZZB93]. Its search space can be set to that of linear, zigzag or bushy trees. We also describe the cost function, which models a shared-nothing parallel execution environment.

2.1 Optimization Context

The DBS3 query compiler takes as input queries expressed in ESQL, a conservative extension of SQL supporting objects and recursion [GV92]. In this paper, we consider only select-project-join queries. The query optimizer selects the best parallel execution plan. The underlying execution system can be either a shared-nothing or a shared-memory multiprocessor [BCV91]. In this paper, we restrict ourselves to shared-nothing for it is the most general case.

Execution plans are modelled by PTs. A PT is a labelled binary tree where the leaf nodes are relations of the input query and each non-leaf node is an operator node (e.g., join, union) whose result is a *transient* relation. A join node captures the join between its operands. The execution aspects, such as the join algorithm, are expressed by means of *execution annotations*. Suppose the following SQL query:

> **Select** * **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

Figure 1 shows four PTs for this query. We say that a PT is *complete* if it captures all the relations of the input query (e.g., all the PTs in Figure 1 are complete).

Our optimizer explores both inter-operation and intra-operation parallelism. ***Inter-operation*** parallelism can be dataflow or independent. ***Dataflow*** parallelism is due to ***pipelining***, i.e., one operation starts as soon as one tuple of the operand is produced. Otherwise, the operand is ***stored***, i.e., the corresponding transient relation is entirely produced before it is consumed by the next operation. PTs are enriched to express pipelining (resp. storing) of transient relations, through directed (resp. undirected) arcs. For the PTs shown in this paper, we adopt the following convention: the operand consumed in pipe is the right input of a node, and the stored operand is at the left side.

[SD90] studied two formats for scheduling linear plans: left-deep and right-deep trees. In a left-deep tree, all transient relations are stored, whereas in a right-deep tree they are consumed in pipeline. Thus all nodes of a right-deep tree execute in parallel whereas in a left-deep tree, only one node executes at a time. For example, Figure 1.(ii), shows a right-deep tree where j4, j5 and j6 are executed in pipeline, and Figure 1.(i), shows a left-deep tree where the execution of j2 starts only after j1 is completed and its result stored. Left-deep and right-deep are two extreme ways to schedule linear execution plans. In [ZZB93], we proposed a new scheduling strategy called zigzag. In a zigzag tree, the transient relation may be either stored or pipelined. Zigzag trees are an intermediate format between left-deep and right-deep trees. That is, they allow to slice an execution plan into sub-trees scheduled as right-deep trees and executed in sequence. It is different from the slicing strategy proposed in [SD90] in the way the result of a sub-tree is consumed by the first operation of the subsequent sub-tree. Figure 1.(iii) shows a zigzag tree where j8 and j9 are executed in pipeline, but j8 starts

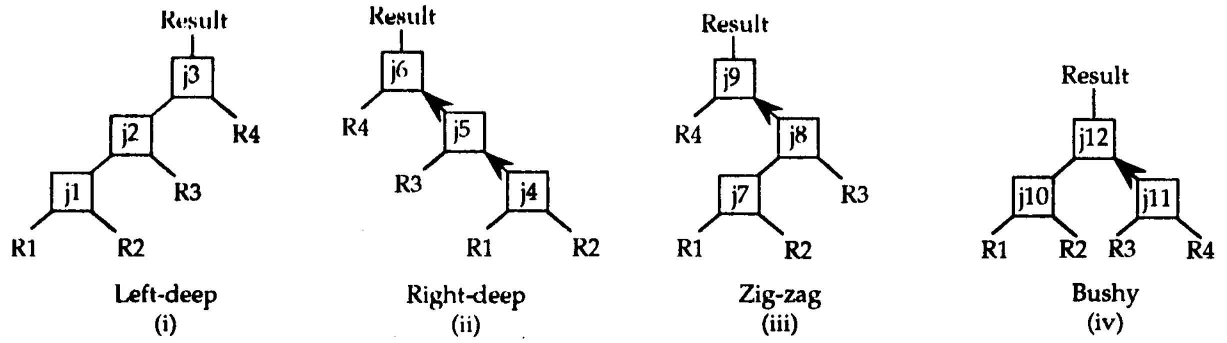


Figure 1: Execution Plans as Processing Trees

only after j7 has completed. Bushy PTs, as the one in Figure 1.(iv), enable another type of inter-operation parallelism, called *independent*, because the operations are independently executed. For example, j10 and j11 are independently executed, because they do not involve the same data.

Pipelining and storing indications split the PT into non-overlapping sub-trees, called *phases*. Pipelined operations are executed in the same phase, whereas a storing indication establishes the boundary between one phase and a subsequent phase. Thus, the execution *scheduling* of a PT can be deterministically derived. The reasons for splitting the PT into phases are twofold. First, the algorithm that implements a PT operation may require the storing of one of its operands (e.g., the hash join algorithm). *Resource contention* is another reason for splitting a PT into phases. If a sequence of operations requires more memory than available to execute simultaneously, it is split into one or more phases. For example, the PT in Figure 1.(iii) is split into two phases, although it could be executed in a single phase if enough memory were available.

Intra-operation parallelism means that a PT operation is executed simultaneously by several nodes. We call *node* one processor, together with its memory and disk. To execute an operation in an intra-parallel manner, the operands must have been previously *partitioned*, i.e., horizontally fragmented, through the nodes[1]. The set of nodes which store a relation is called its *home*. For example, the home of relations R_1 and R_2 is h_1, and the home of relations R_3 and R_4 is h_2. Once R_1 and R_2 are placed on the same home h_1, and the partitioning function is based on their join attribute A, the join j1 can be executed in an intra-parallel fashion on h_1. However, to execute j2, either the result of j1 has to be repartitioned on h_2, or R_3 on h_1. The *home of an operation* is the one in which both operands are located (e.g., the home of j1 is h_1). Usually, the optimizer tries to exploit the placement of the operands to avoid repartitioning. However, execution plans where both operands are repartitioned to increase the degree of parallelism need also be investigated. PT nodes must bear execution annotations to indicate repartitioning, when needed, as well as the algorithm that implements the operation (nested loop or hash).

[1]The way a base relation is partitioned is a matter of physical design.

2.2 Search Space Size

In this section, we investigate the size of the different search spaces. The analysis uses standard combinatorial methods, so details are omitted. Given a query with n relations, the question is how many PTs can be built within each type of search space, without considering several join methods or scheduling, i.e., only the PT shape matters[2]. The results are summarized in Table 1.

First, consider a bushy space. The number x_n of possible PT shapes for a query of n relations, supposing that each PT root node has a left (resp. right) operand capturing k (resp $n-k$) relations, is

$$x_n = \sum_{k=1}^{n-1} \binom{n}{k} x_k x_{n-k}, \textit{where} \qquad x_1 = 1$$

The solution is proven to be $x_n = (2n-2)!/(n-1)!$ in [TL91], which computes the maximum number of PT shapes, obtained when Cartesian products are permitted or all relations are pairwise joinable.

For chain queries (each relation can join with at most two others, and the two "ends" can join with only one) without Cartesian products, there are fewer bushy trees, computed by y_n below,

$$y_n = \sum_{k=1}^{n-1} 2 y_k y_{n-k}, \textit{where} \qquad y_1 = 1$$

as the chain can be partitioned in $n-1$ places, but either partition may be the left child. The relationship

[2][TL91] computes upper and lower bounds for the search space, supposing that the optimizer produces many PTs with the same shape, considering the home of the transient relations and the scheduling.

$y_n = 2^{n-1}x_n/n!$ may be verified by substitution in the recurrence. Chain queries have the fewest possible plans, so y_n gives the minimum number of bushy PT shapes.

Now consider a space of linear PTs. If all relations are joinable or Cartesian products are investigated, we have $n!$ as the maximum number of PT shapes in a left or right-deep space, because all permutations of the n relations are considered. In a zigzag search space, the maximum number is multiplied by 2^{n-2}, because, in each point of a permutation, except the first and second points, the base relations can be joined as the left or right operand. What is a little surprising is that the number z_n of linear PTs is exponential even for chain queries. If there are k operations to the left of the one chosen to be performed first, then there are $\binom{n-2}{k}$ ways to complete the plan (and there are two choices for the chosen first operation). This gives the formula

$$z_n = \sum_{k=0}^{n-2} 2\binom{n-2}{k} = 2^{n-1}$$

which is the minimum number of left or right-deep PT shapes. The minimum number of zigzag PT shapes is obtained by multiplying by 2^{n-2}, as for the maximum number.

2.3 Exploring the Search Space

The search space is explored by the optimizer search strategy, which either builds PTs or modifies complete PTs. When building PTs, the main optimizer action is **expand**(p), which generates new PT nodes by joining one relation (base or transient) to the PT node p . Besides requiring that the relation to be joined with p is not yet captured by p, the implementation of **expand**(p) depends on the type of the search space. For example, in a linear search space, the relation to be joined must be a base relation, while in a bushy space it may be a base or a transient relation. Action **expand** may generate several *successor* nodes to a PT node. It is up to the search strategy to keep some or all of them [LV91].

A randomized search strategy builds one or more *start* solutions and tries to improve them by applying random transformations . The basic action in such strategies is **transform**(p), which applies some transformation to a complete PT p. Only transformations that produce another complete PT in the same search space are applied. These are called *valid* transformations. For example, if the optimization search space of p is such that Cartesian products are not investigated, then a valid transformation must not produce a PT with a Cartesian product . The valid transformations are also conditioned by the shape of the investigated PTs:

- in a *left-deep* space, valid transformations are the *left join exchange* [IC91], and the *swap*, which chooses randomly two points in the current PT and swaps the base relations consumed by them [Sw89];
- in a *right-deep* space, the valid transformations are the *right join exchange* (join(R_1, join(R_2,R_3)) $\rightarrow$ join(R_2, join(R_1,R_3))) and the swap;
- in a *zigzag* space, a valid transformation is one above and the *join commutativity* (join(R_1,R_2) $\rightarrow$ join(R_2,R_1))
- in a *bushy* space, valid transformations are the *join exchange*, the *join commutativity* and the *join associativity* (join (join(R_1, R_2), R_3) $\rightarrow$ join(R_1, join(R_2,R_3)))

The optimizer search strategy is responsible for guiding the application of **expand**(p) and **transform**(p), by choosing p at each step, deciding on the successors to keep, stopping the search and so on. In other words, it is responsible for deciding how many points of the search space are investigated and in which order [LV91]. The choices are based on a cost metrics described in the next section.

2.4 Cost Model

In this section, we propose a cost model that captures all the aspects of parallelism and scheduling. In order to compare PTs, we discuss the drawbacks of previous proposals which do not deal with scheduling.

2.4.1 Comparing PTs

Our optimizer explores both kinds of parallelism, intra and inter-operation. The latter involves the scheduling of execution plans. We distinguish two types of inter-operation parallelism: dataflow and independent. However scheduling execution plans that allow only dataflow parallelism (linear plans) is easy. Independent parallelism (as in bushy plans) is more involved because there are more scheduling alternatives.

During the exploration of the search space, the search strategy compares PTs with respect to their cost. Greedy search strategies, that build PTs by depth-first search, or randomized ones, that transform complete PTs, keep at most one PT after each action (i.e., expand or transform). In these cases, only the cost estimate is relevant when comparing a PT with another one. On the other hand, a Dynamic Programming (DP) strategy [Se79] builds PTs by breadth-first, keeping all incomplete PTs that are likely to yield an optimal solution. Suppose *PTspace* is the set of all PTs (complete or incomplete) built by the optimizer search strategy at some point. At each expand, the search strategy discards from *PTspace* the most expensive PT

Search Space	Min.[a] # of PT shapes	Max.[b] # of PT shapes	Max. # of PT shapes for 5 relations	Max. # of PT shapes for 10 relations
Left or Right-deep	2^{n-1}	$n!$	120	3,628,800
Zigzag	$2^{n-2}.2^{n-1}$	$2^{n-2}.n!$	920	232,243,200
Bushy	$\frac{2^{n-1}}{n}\binom{2n-2}{n-1}$	$\frac{(2n-2)!}{(n-1)!}$	1,680	17,643,225,600

[a]Chain queries, PTs without Cartesian products.

[b]Queries where all relations are pairwise joinable or PTs with Cartesian products.

Table 1: Maximum and minimum number of PT shapes in different search spaces.

nodes among the "equivalent" (*equiv* for short) ones, i.e.,

$$(\forall p, p\prime \in PTspace)\ \ (equiv(p,p\prime)) \wedge (cost(p) > cost(p\prime)) \Rightarrow PTspace := PTSpace - \{p\}$$

The cost estimate of a PT takes into account its schedule. Thus, it may happen that, given two schedules for p and $p\prime$ and a PT q,

$$cost(p) > cost(p\prime) \text{ and } cost(Join(p,q)) < cost(Join(p\prime,q))$$

Therefore, it is better not to discard PT node p when compared to $p\prime$. In other words, the equivalence criterion must include the comparison of plan scheduling. This problem is similar to the tuple order of the result of a given PT in System R [Se79], and to resource consumption in [GHK92]. In the former, the order information influences the costs of subsequent merge joins, GROUP-BY and ORDER-BY clauses. More generally, all the aspects of a PT that affect cost estimation, and which may favor successor nodes, must be considered when comparing PTs to discard the most costly ones.

The equivalence criterion abstracts all the properties of a PT that are used by the cost model: the set of captured relations, the home of the produced transient relation, and the scheduling. More formally:

$$(\forall p,p\prime \in PTspace)\ ((rels(p) = rels(p\prime)) \wedge (home(p) = home(p\prime)) \wedge (sched(p) = sched(p\prime)) \Rightarrow equiv(p,p\prime))$$

The formulas for cost estimation take into account the used machine resources (e.g. processors), the number and the structure of the phases which define the schedule of the execution plan. Consequently, the criterion *sched* is refined as follows:

$$(sched(p) = sched(p\prime)) \iff ((numberOfPhase(p) = numberOfPhase(p\prime)) \wedge (\forall i \in phase(p)\ usedResources(p,i) = usedResources(p\prime,i)))$$

Based on this definition, linear PTs scheduled as right-deep trees (thus executed in one phase) and consuming the same resources (i.e., executed on the same home) are considered equivalent. On the other hand, bushy plans, scheduled in more than one phase, are not equivalent to plans scheduled as right-deep trees.

Our proposal may be seen as an extension to [GHK92] by adding another dimension to the resource vector to take into account the execution over more than one phase. The cost metrics becomes a matrix, where each row represents the resource consumption over one phase. If the plan is scheduled as a right-deep tree, we obtain a vector as in [GHK92].

card(R)	number of tuples in relation R
width(R)	size of one tuple of relation R
cpu	CPU speed
network	network speed
packet	the size of a packet
send	the time for a send operation
receive	the time for a receive operation

Table 2: Cost Model Parameters

2.4.2 Cost Functions

In this section, we define the cost estimate of a PT containing only join nodes. All formulas given below compute response time and we simply refer to it by *cost*. In addition to the traditional assumptions (uniform distribution of values and independence of attributes), we also assume that the tuples of a relation are uniformly partitioned among the nodes of its home, and there is no overlap between nodes of different homes, although several relations may share the same home.

In the following, R refers to a base relation of the physical schema, and N to the operation captured by a PT node. p denotes, in the same time, a PT and the transient relation produced by that PT. The parameters (database schema or system parameters) used in the cost model are shown in Table 2.

An optimal execution of the *Join* operation, requires each operand relation to be partitioned the same way. For example, if p and q are both partitioned on n nodes using the same function on the join attribute, the operation $Join(p,q)$ is equivalent to the union of n parallel operations $Join(p_i,q_i)$, with $i = 1,n$. If the afore mentioned condition is not satisfied, parallel join algorithms [VG84] attempt to make such condition available by reorganizing the relations, i.e., dynamically repartitioning the tuples of the operand relations on the nodes using the same function on the join attribute.

We first estimate the cost of repartitioning an operand relation R. Obviously, if the relation is appropriately partitioned, this cost is 0.

Let $\#source$ be the number of nodes over which R is partitioned, and $\#dest$ be the number of nodes of the destination home. Each source node contains $card(R)/\#source$ tuples, thus it will send $card(R) * width(R)/(n * packet)$ packets. If we assume that tuples will be uniformly distributed on destination nodes, then each node will receive $card(R)/\#dest$ tuples, and thus will process $card(R) * width(R)/(m * packet)$ incoming packets. Since a destination node starts processing only when the first packet arrives, the cost of repartitioning R on $\#dest$ nodes is,

$$\begin{aligned}&cost(part(R)) = \\ &max((card(R) * width(R)/(\#source * packet)) * send, \\ &\quad (card(R) * width(R)/(\#dest * packet)) * receive \\ &\quad + send + packet/network)\end{aligned}$$

The cost of joining tuples of PTs p and q, where p and q are respectively the pipelined and stored operands of the $Join$ operation, is

$$\begin{aligned}cost(Join(p,q)) = &max(cost_{alg}(Join(p,q)), cost(part(p))) \\ &+ cost(part(q))\end{aligned}$$

where $cost_{alg}(Join(p,q))$ is the cost to process the join at one node. It depends on the join algorithm used [Za90]. The repartitioning of p is performed simultaneously to the join processing, after the repartitioning of q has completed.

Given a PT p scheduled in phases (each denoted by ph), the cost of p is computed as follows

$$\begin{aligned}cost(p) = \textstyle\sum_{ph \in p}(&max_{N \in ph}(cost(N) + pipe\_delay(N)) \\ &+ store\_delay(ph))\end{aligned}$$

where $pipe\_delay(N)$ is the waiting period of node N, necessary for the producer to deliver the first result tuples. It is equal to 0 if the input relations of N are stored. $store\_delay(ph)$ is the time necessary to store the output results of phase ph. It is equal to 0 if ph is the last phase, assuming that the result are delivered as soon as they are produced.

To study the behavior of search strategies, an important property of the cost function is the *Adjacent Sequence Interchange* (ASI) property [MS79]. Consider a PT whose nodes can be mapped to a sequence of relations of the form AUVB, where A, B are arbitrary sequences and U, V non-null sequences. [KBZ86] shows that the ordering of U and V in the PT is purely based on the properties of the sequences U and V, and can be decided irrespective of the rest of the sequence (i.e. A and B), only if the cost function verifies the ASI property. In our cost model, some characteristics of the result of a PT node may have an impact on the cost and the optimization choices performed for the subsequent PT nodes. For example, the home of a transient relation impacts for the cost of the next consuming operation. Therefore, choosing between two orderings of sequences in a PT is dependent on the rest of the sequences. The ASI property is, then, not verified by our cost model. The impact of this information on randomized strategies will be studied in Section 3.2.

3 Search Strategies

The optimizer *search strategy* is the search algorithm which investigates PTs in a given search space. It may be either *deterministic* (i.e., builds PTs in a deterministic fashion) or *randomized* (i.e., investigates new PTs by applying random transformations on PTs). In the DBS3 optimizer, we have implemented the extensible approach described in [LV91] with several search strategies, which we now describe.

3.1 Deterministic Search Strategies

Deterministic strategies always choose the same PT for a given query in a given search space. Examples of such strategies are *Greedy* and *DP* [Se79].

Greedy strategies are the faster ones because they investigate very few PTs. They proceed by depth-first, starting from the relation with the least cardinality. After each expansion, all the successor nodes are pruned but one. Different heuristics may be used in conjunction with a Greedy strategy. In the DBS3 optimizer, we used the Augmentation Heuristic [Sw89]: the plan starts by the relation with the least cardinality and, at each expansion, the successor node with the least cost is retained. Greedy strategies are also used to generate start solutions for randomized strategies.

We have studied a variant that produces what we call the *Uniform Greedy* solution. This strategy produces one complete PT starting from each base relation via the Augmentation Heuristic, then chooses the least costly among these complete PTs as the solution. We have observed that this is a very effective strategy to find a start solution for Simulated Annealing in linear spaces.

DP proceeds by breadth-first, selecting the least recently generated PT node to be expanded at each step. The strategy is almost exhaustive, because heuristics are used to prune *bad* states as soon as possible. Pruning discards PT nodes which are "equivalent" to some other existing PT node with higher cost (see Section 2.4.1 for a discussion on the equivalence criterion). Because the pruning policy considers the homes, the number of phases and the resource consumption of each phase, the number of investigated PTs is much larger than in a centralized environment. The main resource constraint when using DP proved to be the amount of space used by the optimizer. Because of breadth-first search, many incomplete PTs are generated before the search ends. Thus, running DP on reasonable queries (e.g., 8 relations or more) in a bushy search space often causes the optimizer to run out of space. This is the motivation to use randomized strategies.

3.2 Randomized Strategies

Randomized strategies concentrate on searching the optimal solution around some particular points. They do not guarantee that the best solution is obtained, but avoid the high cost of optimization. First, one or more *start* PTs are built by a Greedy strategy. Then, the algorithm tries to improve the start PT by visiting its *neighbors*. A neighbor PT′ is obtained by applying a **transform** to a PT, i.e., PT′ = **transform**(PT). A PT is a *local minimum* if it has the least cost among all its neighbors. A PT is a *global minimum* if it has the least cost among all the local minima. The effectiveness of a randomized strategy is related to its ability of reaching the global minimum. This is more difficult as there are more local minima.

3.2.1 Randomized Strategies vs. Cost Metrics

The number of local minima depends on the execution space as captured by the cost model. Thus, the behavior of randomized strategies is significantly affected by the properties of the cost metrics. [IC91] stated that in search spaces following the ASI property, there is a unique local minimum, which is the global minimum. This enables very effective optimization. After choosing a randomly generated start PT, apply **transform** actions until no less costly neighbor PT is generated. In centralized optimizers, cost functions corresponding to some join algorithms (e.g., merge join, hash join) do not follow the ASI property. Nevertheless, [IC91] observed that many subspaces follow ASI, leading to many local minima. This makes randomized strategies investigating many such subspaces to increase the opportunities of finding the global minimum. Each strategy implements this investigation of several subspaces in a different way.

When comparing the cost metrics of a centralized optimizer to ours, where scheduling is taken into account, two observations can be drawn. First, the cost distribution is much more scattered, implying the need to look into more subspaces than in the centralized case. Second, the cost of a neighbor PT can be radically different (up to a factor of 10) from that of the starting PT[3]. This implies that, in a given subspace, the optimizer may be able to reach a local minimum earlier, with less **transforms** than in the centralized case. Both factors are more prevailing in bushy execution spaces, where there are more choices for scheduling, than in a linear one.

These observations lead to an important conclusion. A parallel optimizer acting in a bushy space should investigate more subspaces than a centralized one, spending less time in the exploration of each subspace. In the sequel, we present the randomized strategies implemented in the DBS3 optimizer which take into account these observations.

3.2.2 Iterative Improvement (II)

II [Sw89] performs several *runs*. Each run consists of improving one start PT by applying **transforms**, until a local minimum is reached. Thus, only **transforms** that generate PTs which are less costly than the original one are accepted. The number of runs is equal to the number of relations in the query, i.e., each start PT is greedily generated from a different base relation. As the strategy is not exhaustive, not all the neighbors are visited and it is difficult to recognize a local minimum. A bound on the number of visited neighbors is typically accepted as a criterion for reaching the local minimum. This bound is proportional to the query size: a parameter *localBudget* times the number of relations in the query.

II visits as many subspaces as the number of relations in the query, due to the choice of the several start PTs. This has been proved effective in centralized optimizers [Sw89], and we will show that it is also the case in linear parallel execution spaces, in Section 4.4. However, the number of visited local minima is insufficient in a parallel bushy space, due to the excessive scattering of the cost distribution. This explains the low quality of the plans chosen by II running in a bushy space. The localBudget parameter may be smaller in bushy spaces. II converges very quickly to each local minimum. The reason is that, in bushy spaces, a single **transform** may generate a neighbor PT with a very low cost, compared to that of the original one.

3.2.3 Simulated Annealing (SA)

Contrary to II, all the runs of SA are performed over a single start state. However, the system has a temperature property *temp*, which is reduced at each run. Optimizer parameters allow specifying the initial temperature (a factor multiplied by the cost

[3]This is not the case in centralized bushy spaces [IC91].

of the start PT, typically 2.0), and the decreasing ratio. The algorithm stops when temp $\leq$ the "freezing" temperature, or when a "stable" solution has been found (i.e., it stays unchanged during four runs, as proposed in [IC91]). As in II, the local budget for each run is related to the size of the problem. The criterion for accepting a **transform**, however, is different, because transformed PTs with a higher cost than the original PT are accepted with some probability, that decreases with the temperature. Accepting *bad* moves corresponds to a "hill climbing" [IC91]: on the other side of the hill there may exist a better solution.

We have observed that the augmentation heuristic often approaches the optimal linear solution provided that it has been applied to a "good" start relation. It is not possible to predict at first sight which is the "good" start relation. Similar to [Sw89], we propose to choose the *Uniform Greedy solution* (see Section 3.1) as the unique start solution for SA in linear spaces. This increased significantly the quality of the solution chosen by SA.

During the experiments, we observed that SA performed very well in linear spaces, but often failed to choose the optimal solution in bushy spaces, even when Uniform Greedy was chosen as the start solution. The reason is that a single start solution provides too few chances for exploring a large search space, e.g., with bushy trees. The standard way to do this, i.e., through the acceptation of "bad" moves, proved to be not sufficient in a parallel bushy execution space. This motivated our proposal of a modified version of SA.

3.2.4 Toured Simulated Annealing (TSA)

TSA performs n tours of SA, each tour starting with a different Greedy solution, where n is the number of base relations in the query. Each start solution is obtained as for II, i.e., it is greedily built by the augmentation heuristic beginning at each base relation.

To prevent the optimization cost of increasing excessively, each tour is given a very low ratio for the initial temperature (0.1 instead of 2.0), impelling the system to accept bad moves less often. This is somewhat analogous to the 2-Phase Optimization strategy proposed in [IC91]. We also introduced an optimization budget, equally divided by the number of tours, that limits the optimization effort. Similarly to what happened in II, we observed that each "tour" reached a local minimum very early, and spent the remaining time exploring useless solutions. As we keep track of the best solution found so far, this "wondering" is not harmful from the point of view of quality of the chosen plan, but increases uselessly the optimization cost. So, each tour is given a limited optimization budget, based on the number of generated PT nodes. We show in Section 4 that TSA presents the best trade-off between the optimization cost and the quality of the execution plan.

3.2.5 Randomized Strategies vs. Transform Actions

The behavior of randomized strategies is affected by the nature of the applied **transform** actions. Recall that a transformation action must be *valid*, i.e., constrained in the considered search space. All of the described transformations generate a PT within the same search space as the original one with regards to their shape (see Section 2.3). Often swaps and join exchanges turn out to be invalid if the search space prevents join permutations that imply Cartesian products. As these transformations are randomly chosen, their *validity* must be verified before they are applied. The number of successful attempted actions is related to the "connectivity" of the query (i.e., the number of join predicates connecting the relations).

As the join exchange is a particular case of swap, we conducted some experiments to study the behavior of randomized strategies using one or the other. If only **joinExchange** is used, it may require $\binom{n}{2} \approx \frac{1}{2}n^2$ such actions to reach the optimal plan; at each step there are $n-1$ possible choices. On the other hand, with **swap** the target can always be reached by *some* sequence of $n-1$ or less actions, and at each step, there are $\binom{n}{2}$ choices. In some sense, **joinExchanges** are "small" moves, while general **swaps** constitute "large" moves.

Based on these considerations, starting from a plan that is well away from the optimal, we would expect **joinExchanges** to produce small improvements, but with a higher success rate, compared with **swap**. The initial progress should be more remarkable with **swap**, sometimes being very lucky and sometimes quite unlucky. As a local minimum is approached, **swap** will have an even higher failure rate, as large moves tend to be unproductive.

On bushy spaces, the validity of **transform** actions are less sensitive to the connectivity of the query. As their effect is localized on one point of the PT, there is a lesser (resp. none) probability that a join exchange or a join associativity (resp. join commutativity) will fail to produce a valid PT in a search space avoiding Cartesian products. Experiments where join exchange was not available for bushy spaces performed poorly. Although the effect of a join exchange can be produced by a combination of join associativity and join commutativity, it is essential to have it as an additional transform action, because it may be valid when a join associativity may not.

4 Measurements

We now measure the trade-off between the optimization cost versus the quality of the produced execution plan. In order to make meaningful measurements, we provided the optimizer with "sensors" and "buttons" to abstract implementation details and to grasp the essence of the optimizer behavior. We investigate many combinations of search spaces and search strategies for some sample queries. We show that optimization becomes intractable in some situations (e.g., an exhaustive strategy applied to 9-way join queries in a bushy search space). We demonstrate that restricting the search space may lead to miss the optimal plan. As an alternative to reduce the optimization cost, we use randomized strategies. We show that they perform well on parallel environments and choose the optimal plan in most cases.

4.1 Testbed

Our measurements use a realistic application: the Portfolio Club Experimental Model (PEM). This was designed to provide a realistic experimental base for complex query definition, evaluation and benchmarking on the DBS3 system [JKK90]. PEM constitutes a simplified model for an application on share market and investment/portfolio management. The PEM schema contains 20 relations joinable through foreign keys. There are three kernel relations, "enterprises", "investors" and "holdings", to which disjoint sets of the remaining relations are joinable. Besides, "enterprises" and "investors" are joinable, as well as "enterprises" and "holdings".

The testbed catalogs were generated automatically using three key parameters which correspond to the cardinalities of the kernel relations. The cardinalities of relations vary with respect to each other considerably, as in real applications. The catalog describes also the partitioning of relations, the number of nodes and the attributes used by the partition function. Disjoint subsets of the relations were partitioned on three disjoint homes, containing one kernel relation plus the set of relations whose foreign key is the primary key of the kernel relation. We do not consider indexes. The machine consists of 30 nodes (i.e., processors) and each home is constituted by 10 nodes. Table 3 gives the values of the machine parameters used by the cost model, validated on the current EDS shared-nothing parallel system.

We conducted our experiments using equijoin queries ranging from 4 to 12 relations. If we consider the database schema as a graph, where a node corresponds to a relation, an edge connects two nodes if the corresponding relations share the same attribute name. Thus, a query on a given subset of relations is the subgraph containing them. We considered all the "implied" predicates in a query (e.g., R1.A = R2.B and R2.B = R3.B implies R1.A = R3.B). Due to the nature of the catalog, "star" queries are frequent. They are generally considered as the hardest to optimize, because of the large number of permutations of relations not involving Cartesian products. Each query was optimized with respect to three catalogs, each containing the same PEM schema, but with widely varying statistics. The experiments with the optimizer were conducted on a SUN-4 workstation with a 16 MIPS CPU and 32 Mbytes of memory.

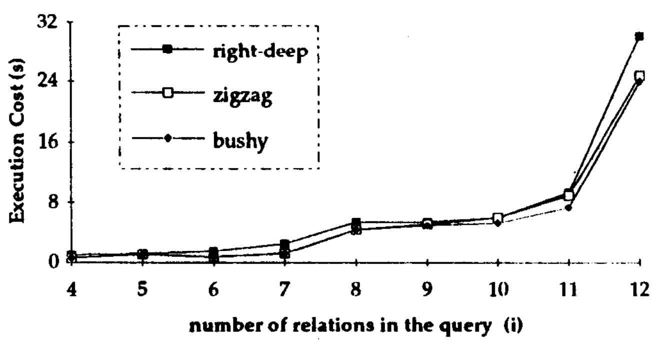
number of MIPS per CPU	30 MIPS
speed of the network	200 Mbits/second
size of a packet	512 bytes
time for a send operation	33 μs
time for a receive operation	23 μs

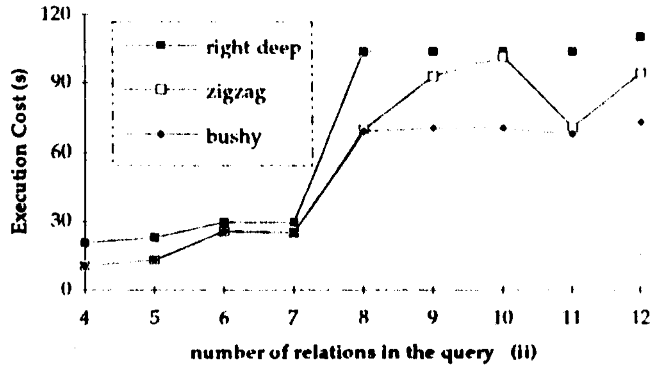
Table 3: Values of Cost Model Parameters

4.2 Experimentation Methodology

The effectiveness of a strategy is strongly related to the consumed resources, typically time and space. A common difficulty with resource measurement is that it is very implementation dependent. Therefore we attempted to identify characteristic actions used by the various strategies, and to count them, rather than rely on CPU time or memory size alone. For deterministic strategies, **expand** is the most typical action, while **transform** is typical for randomized ones. As both actions generate new PT nodes, we use the counter of *generated PT nodes* to compare different strategies. Recall that, if randomized strategies are used within a search space with no Cartesian products, many transformation attempts may fail before getting to a "successful" one. The failures are also tracked by the counter of generated new PT nodes. Thus, the number of generated *new PT nodes* correlate well with time and space. The number of generated new PT nodes per CPU second was 20 for both DP and randomized strategies. Besides its use for experimental measurements, this counter provides an implementation-independent way to constrain the resources consumed by the optimizer. The cost of a PT represents the response time as computed by the cost functions of the optimizer.

The optimizer behavior depends on a parameter file, that fixes the search space (right-deep, zigzag, and bushy. Left-deep are not considered in the experiments) and the search strategy (DP, II, SA, and TSA), and parameters for randomized strategies (e.g., global and local budgets, ratios for initial temperature and temperature decrease for SA and TSA).





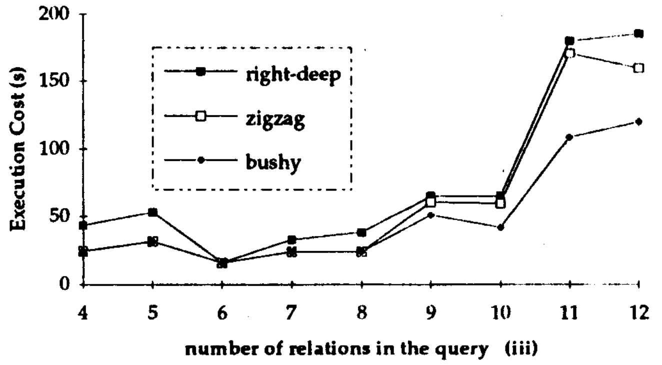


Figure 2:
Execution cost of optimal plans obtained by DP/TSA for varying number of relations in the query. The three graphs correspond respectively to (i) *catalog*(1100,500,1100), (ii) *catalog*(1100,5000,1100), and (iii) *catalog* (10000,500,10000).

4.3 Quality of Query Execution Plan vs. Search Space

In the first set of measurements, we were interested in verifying if choosing a larger search space changes the optimal solution. We applied DP to several queries in different search spaces using three different catalogs referred to by $catalog(i, j, k)$ where i, j, and k are the cardinalities of "enterprises", "investors", and "holding" respectively.

Figure 3 shows the cost of the execution plans obtained for varying number of relations in the query in right deep, zigzag and bushy search spaces. The zigzag solution is always better than the right-deep one. The bushy solution is sometimes better than zigzag. Clearly, the choice of a larger space enables finding a better plan. In large spaces, the optimizer is able to investigate all possibilities that lead to the best parallelization, regardless of the initial partitioning of relations.

DP becomes intractable, i.e. runs out of memory, in zigzag or bushy spaces for queries with 9 relations or more[4]. In these cases, we used the cost of the solution obtained by TSA. The good surprise is that TSA in a zigzag or bushy spaces was able to find better solutions than DP running in a right-deep space. In fact, our realistic testbed is such that subsets of relations are located on disjoint homes. In this situation, independent parallelism is better than dataflow, which incurs time-sharing. This favors zigzag and bushy PTs.

[4]Recall that the queries are very connected by join predicates, due to the implied predicates.

4.4 Quality of Query Execution Plan vs. Search Strategy

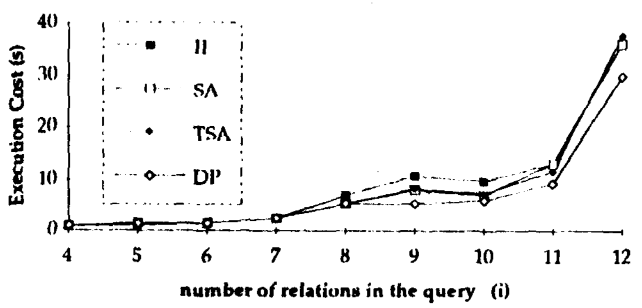
We applied II, SA and TSA to the same queries as before, using the catalog(1100,500,1100), in linear and bushy spaces.

Figure 3 shows the execution cost of solutions chosen by randomized strategies in linear and bushy spaces.

In a linear space, all strategies obtain the same plan or near as a final solution in most cases (see Figure 3(i)). Moreover, the cost of the chosen plan is equal or near to that chosen by DP, when the comparison is possible.

In bushy spaces, for queries with 7 relations or less, the chosen solutions have the same cost for all strategies too (see Figure 3(ii)). However, for more complex queries, the cost of bushy solutions chosen by different randomized strategies are considerably different. As explained in Section 3.2.1, this is due to the fact that the cost distribution in bushy parallel spaces is very scattered. As TSA is able to explore more points in the search space than the other strategies, it is able to find better bushy plans. It is, then, the best choice when DP is no more feasible.

Randomized strategies have not been previously proposed to parallel optimization. Our experiments showed that they are very effective, specially with some improvements. Both modifications proposed to SA (i.e.,



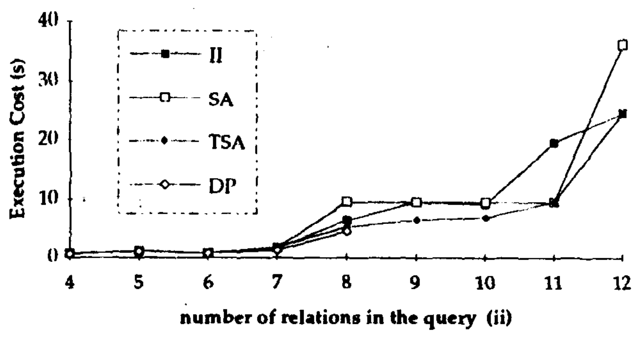


Figure 3: Comparison of Randomized Strategies: (i) right-deep, (ii) bushy.

the Uniform Greedy as the start solution for linear spaces and the Toured version) have the same goal. They give this strategy a better chance of looking around for more points in the solution space.

4.5 Trade-off between Optimization and Execution Costs

We measured the trade-off between optimization and execution costs in right-deep and bushy spaces, using several strategies. The trade-off expresses the increase rate in both the optimization and execution costs incurred by a strategy, within a given search space for a given query. Summing both rates corresponds to consider the total resources consumed by optimization and execution. Each increase rate is estimated with respect to the best cost found so far by some strategy. Given a strategy S, a space E, and a query Q, the trade-off of strategy S to optimize Q in space E is,

$$trade{-}off(S,E,Q) = IR\_OC(S,E,Q) + IR\_EC(S,E,Q)$$

where IR_OC and IR_EC are the increase rate in optimization cost and execution cost respectively, i.e.,

$$IR\_OC(S,E,Q) = (OC(S,E,Q) - bestOC(E,Q)) / bestOC(E,Q)$$

$$IR\_EC(S,E,Q) = (EC(S,E,Q) - bestEC(E,Q)) / bestEC(E,Q)$$

$bestOC(E,Q) = min_{\forall St} OC(St,E,Q)$
$bestEC(E,Q) = min_{\forall St} EC(St,E,Q)$
$OC(S,E,Q)$ measures the optimization cost (number of generated new PT nodes) of S to optimize Q in E, and $EC(S,E,Q)$ measures the execution cost of the PT produced by S for Q.

Figure 4 shows the trade-off of DP and randomized strategies for varying number of relations in the query. For small queries, with less than 7 relations, DP offers the best trade-off. For larger queries, the situation is inversed to the benefit of TSA in bushy spaces, and II in right-deep space.

This result validated the use of randomized strategies for parallel search spaces when DP becomes intractable.

4.6 Summary

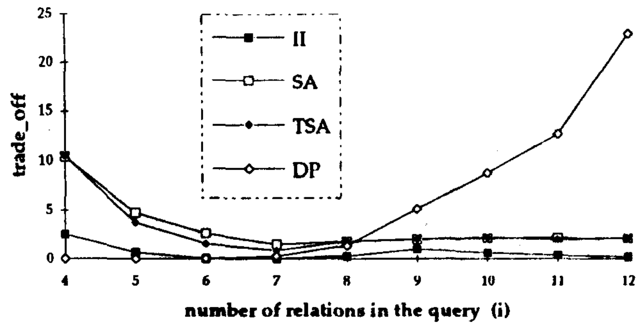
The experiments brought in some important insight regarding optimization in a parallel environment. First, a larger space than right-deep, as zigzag or bushy, provides more chances to find a better plan. With an almost exhaustive strategy in such spaces, such as DP, optimization becomes intractable beyond a certain query complexity. In a parallel environment, this limit is attained even for reasonable queries (8 relations or more). This is due to the fact that the pruning criterion is very restrictive.

In previous work, the problem is dealt with restricting the search space [HS91, SD90]. We chose to apply randomized search strategies instead. For this purpose, we used a modified version of SA, that offers a better trade-off in comparison with other strategies. Our experiments showed that using a randomized strategy within a bushy search space is likely to find a better plan than an exhaustive strategy within a restricted search space.

We did not consider indexes for simplicity. Their impact on our results will be studied in a future work.

5 Conclusion

In this paper, we have studied the trade-off between query optimization and execution costs with DBS3 optimizer. This optimizer explores both kinds of parallelism, intra and inter-operation, through linear or bushy plans, and implements several search strategies including DP, II, and two variants of SA. Using a realistic testbed, we conducted a series of experiments to measure the trade-off between optimization cost and quality of query execution plans.



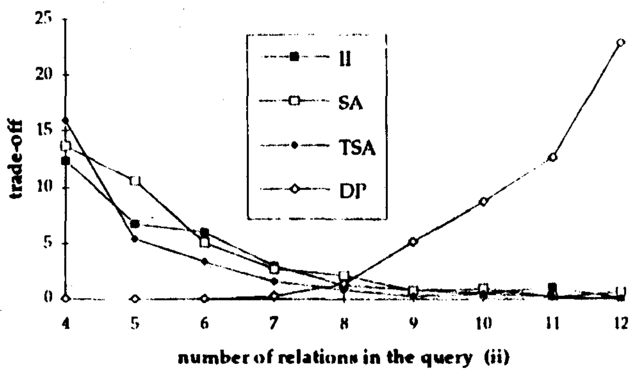


Figure 4: Trade-off between Plan Execution Time and Optimization Cost: (i) right-deep, (ii) bushy.

The main contribution of this paper is to assure tractability through the use of non exhaustive search strategies. For this purpose, we extended randomized strategies for parallel optimization, and demonstrated their effectiveness.

Contrary to previous works, our results show clearly that parallel query optimization should not imply restricting the search space to cope with the additional complexity. We showed that this may lead to missing better plans. It is essential to keep the search space large and to control the search strategy to assure tractability. Combining bushy search spaces with randomized strategies is the best solution, when DP becomes intractable.

References

[BCV91] B. Bergsten, M. Couprie, P. Valduriez: "Prototyping DBS3, a Shared-Memory Parallel Database System", *PDIS* 1991.

[EDS90] EDS Database Group: "EDS Collaborating for a High-Performance Parallel Relational Database", *ESPRIT Conf., Brussels* 1990.

[GHK92] S. Ganguly, W. Hasan, R. Krishnamurty: "Query Optimization for Parallel Execution", *SIGMOD* 1992.

[GV92] G. Gardarin, P. Valduriez: "ESQL2: an Extended SQL2 with F-logic semantics", *IEEE Data Engineering* 1992.

[HS91] W. Hong, M. Stonebraker: "Optimization of Parallel Query Execution Plans in XPRS", *PDIS* 1991.

[IC91] Y.E. Ioannidis, Y. Cha Kang: "Left-deep vs. bushy trees: an Analysis of Strategy Spaces and its Implications for Query Optimization", *SIGMOD* 1991.

[JKK90] J. Jorgensen, S.M. Kellett, N.C. King: "Portfolio Club Experimental Model", EDS Report EDS.DD.11I.0005, Dec 1990.

[KBZ86] R. Khishnamurty, H. Boral, C. Zaniolo: "Optimization of Nonrecursive Queries", *VLDB* 1986.

[LV91] R.S.G. Lanzelotte, P. Valduriez: "Extending the Search Strategy in a Query Optimizer", *VLDB* 1991.

[LVZ92] R.S.G. Lanzelotte, P. Valduriez, M. Zaït: "Optimization of Object-Oriented Recursive Queries using Cost-Controlled Strategies", *SIGMOD* 1992.

[MS79] C.L. Monma, J.B. Sidney: "Sequencing with series-parallel precedence constraints", *Math. Oper. Res.*, 4 1979.

[SD90] D. A. Schneider, D. J. DeWitt: "Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines", *VLDB* 1990.

[Se79] P.G. Selinger et al.: "Access Path Selection in a Relational Database Management System", *SIGMOD* 1979.

[Sw89] A. Swami: "Optimization of Large Join Queries: combining Heuristics and Combinatorial Techniques", *SIGMOD* 1989.

[TL91] K-L. Tan, H. Lu: "A Note on the Strategy Space of Multiway Join Query Optimization Problem in Parallel Systems", *SIGMOD* 1991.

[VG84] P. Valduriez, G. Gardarin: "Join and Semijoin Algorithms for a Multiprocessor Database Machine", *ACM TODS*, Vol. 9, No. 1, 1984.

[Za90] M. Zaït: "Access Method Selection in a Parallel Database System", *Master Thesis*, University of Paris 6, September 1990.

[ZZB93] M. Ziane, M. Zaït, P. Borla-Salamet: "Parallel Query Processing in DBS3", *PDIS* 1993.