Parallelism and Query Optimization*

Mikal Ziane [1,2], Mohamed Zaït [1], Hong Hoang Quang [2]

INRIA [1]
projet Rodin
B.P. 105
78153 Le Chesnay cedex
Rocquencourt, France

Université Paris V [2]
I.U.T. - Dept. Informatique
143, Av. de Versailles
75016 Paris
France

e-mail: Mikal.Ziane@inria.fr

Abstract

Query optimization for parallel execution is an open problem [DeWi90]. However [Hong91] showed that in the context of XPRS the two-phase hypothesis seems to be valid. In this paper we clarify the domain of validity of this hypothesis which states that "the best parallel plan is a parallelization of the best sequential plan". In order to do this, we first clarify the differences between sequential and execution plans and make precise the decisions to be taken by a parallel optimizer. Our analysis suggests that the two-phase hypothesis is usually not valid in DM. In SM it is true as long as disks are not involved. Finally, we used our parallel optimizer [Zian93] to make several experiments which confirmed some of our conclusions.

1. Introduction

According to [DeWi90], optimization for parallel execution is an open problem, and indeed very few papers address the problem globally, while many focus on some specific aspects. However, it is also sometimes assumed that parallel execution has little impact on query optimization. XPRS [Ston88] is a striking example in which the optimizer is the conventional (centralized) optimizer of POSTGRES. More precisely query optimization in XPRS is realized in two independent phases. The first phase is static and chooses the best sequential plan. The second phase is executed at runtime and parallelizes the best sequential plan. This approach relies on the **two-phase hypothesis** which states that "the best parallel plan is a parallelization of the best sequential plan" [Hong91].

As the apparent contradiction between viewpoints like those of [DeWi90] and [Hong91] suggests it, the impact of parallelism on query optimization is not clear. Query optimization consists in finding an execution plan of the lowest possible cost for a query expressed in a declarative language. This involves three aspects: the **search space** which is the set of candidate execution plans for the query, the **cost function** which gives an approximation of the response time or the resource consumption of a plan, and the **search strategy** which determines which plans are considered and in which order.

* This work was sponsored by the ESPRIT III IDEA project.

In a parallel system the search space is a priori larger if no heuristic simplification is made. This is due to the fact that more scheduling alternatives have to be considered as it will be shown below. The cost function is often more complex, especially in case of inter-operation parallelism, and heuristics are more difficult to find. With a larger search space and less heuristics, traditional search strategies such as dynamic programming can become untractable. The solution we adopted in our prototype [Zian93] was to rely on randomized search strategies such as iterative improvement or simulated annealing [Lanz93].

Giving a general picture of the impact of parallelism on the three aspects of query Optimization is beyond the scope of this paper. Our objective is to clarify in which cases the two-phase hypothesis is valid and in which cases it is not. Intuitively, if all the sequential plans got the same speedup when parallelized, then the two-phase hypothesis would always be valid.

More precisely, let SP be a sequential plan and PP its best parallelization. Suppose that the cost function of a sequential plan is its total resource consumption: seqcost(SP) = W(SP). Suppose now, as it is done in several systems ([Hong91], [Zian93]), that the cost function of a parallel plan PP is given by parcost(PP) = W(PP) + k*T(PP) where T is the response time and k is a weighting factor. Parallelizing a plan introduces some overhead, W(PP) - W(SP), for starting and terminating the processes, for synchronization and because of the communications between the processors (in distributed-memory). If this overhead was proportional to W(SP), if the load was always perfectly distributed among the processing elements (processors, disks ...), and if a fixed number of processing elements was allocated to all the plans, then T(PP) would be proportional to W(SP). In such a case parcost(PP) would be proportional to seqcost(SP) and the two-phase hypothesis would be valid. When such a proportional relation does not hold, it is likely that among all the sequential execution plans some might get more benefit from parallelization than others. It would then be unlikely that sequential optimization and parallelization be independent.

Note that XPRS ([Ston88], [Hong91]) is not very far from this situation. The architecture is shared-memory so that all the processors can be assigned to any join, and load balancing is much easier to achieve than in distributed-memory. Moreover, the search-space is restricted to left-deep trees which only allows intra-operation parallelism[1] and thus load balancing (of a single query) is easily achieved if data are uniformly fragmented. Since data are declustered on all the disks in a range-robin fashion, selections that do not use indices always use all the disks.

In this paper we also take into account distributed-memory architectures and more general tree shapes (e.g. right-deep and bushy trees) which allow inter-operation parallelism. We consider here that optimization reduces to join-ordering and to choosing a scheduling strategy for the candidate execution plans. We assume that queries are pure equi-joins, that no indices are available and, like in [Hong91], that there is enough main memory to consider only hash-based join algorithms. We also assume, like [Hong91], that main memory is above the hash-join threshold [Shap86] so that the join algorithm is always hash-based.

[1]In XPRS, inter-operation parallelism is only used to help balance CPU-bound and IO-bound queries [Hong92].

Finally, we did not take into account the fact that some parameters, such as the amount of available main memory, are usually difficult to know before runtime. We supposed that solutions like choose-plan operators [Grae89] allow us to consider this problem as orthogonal to the problem of optimization. One reason to support this view is that, in theory, the systematic use of choose-plan operators would allow to defer a decision without any restriction of the search space. This is of course an important simplification because choose-plan operators would be very inefficient at runtime in case of many alternatives. Note that using choose-plan operators is very different from adopting the two-phase approach. When the two-phase hypothesis is not valid, a decision taken in the first phase can be sub-optimal if it constrains the choices of the second phase.

The first thing to do to clarify the domain of validity of the two-phase hypothesis, is to highlight the difference between parallel and sequential plans. Until now we have followed the viewpoint of [Hong91] that a parallel plan is a parallelization of a sequential plan. However it is necessary now to distinguish two levels of abstraction. In both sequential and parallel optimizers, join ordering produces abstract execution plans which make no assumption on scheduling. In order to execute an abstract plan, scheduling information must be provided either explicitly by the optimizer or implicitly by the execution system.

In section 2 we give a definition of scheduling. In section 3 we make precise the scheduling decisions to be taken by a parallel optimizer. Then we can analyze the domain of validity of the two-phase hypothesis (section 4) and finally present the experiments we run on our prototype optimizer (section 5).

We used our parallel optimizer ([Lanz93], [Zian93]) which supports both distributed-memory and shared-memory architectures to compare the results obtained with the two-phase approach and the results obtained with a one-phase approach. The results confirm, under the assumptions we made above, the validity of the two-phase approach in shared-memory systems but not in distributed memory systems. Our experiments however did not take IOs into account, only CPU utilization.

2. Scheduling of execution plans

In a typical physical optimizer, one of the most important task is join ordering, whose output is a tree of join operations. We call such a tree an **abstract execution plan**. We call a **concrete execution plan** an abstract plan in which operations have been labelled by an algorithm (hash-join in our case), and which is augmented with enough **scheduling** information to be executed by the target system. We assume that the **atomic actions** to be scheduled are tuple productions and consumptions.

In a sequential system, scheduling is most often implicitly fixed to some default strategy so that the execution system is able to directly execute an abstract execution plan. For example, the execution system might impose that only one operation at a time can be under evaluation. This is actually also the case of a parallel system with no inter-operation parallelism [Shasha91]. Note however that "simulated" pipelining sometimes occurs in sequential systems such as system R [Seli79]. The fact that scheduling information is often implicit in a sequential plan, allows to add parallel scheduling information to it, and thus make it a parallel plan. [Grae90] is indeed able to create parallelism in a sequential plan without modifying the sequential algorithms, just by adding an *exchange* operator.

An operation tree imposes a partial order on the set of tuple productions and consumptions, which reflects the obvious constraint that a tuple must be produced before it is consumed. Execution models usually further constrain the execution of operation trees. For example, most of them limit to one operand the possibility to be consumed in pipeline and most (sequential or parallel) algorithms impose that one operand be completely produced before any tuple of the other operand can be consumed. This is the case for hash-based join algorithms which typically consist of two consecutive phases: build and probe.

Definition
Let P be an abstract execution plan and A(P) the set of atomic actions of P. Let TS(P) be the subset of A(P)xA(P) for which (a1,a2) is in TS(P) if and only if a1 and a2 are not performed by the same operation. We define a **scheduling** of P as a subset of TS(P).

Note that a scheduling S of P defines a partial order on the set of atomic actions of P, that is (a1,a2) is in S when a1 < a2, which means that a2 cannot start before a1 is finished. If two actions are not comparable it means that they may be executed in parallel. Scheduling is not concerned by the order of the actions of each operation considered individually. The algorithm of each operation determines the order of the tuples it produces, possibly depending on the order of the tuples it gets. Of course, for a sort operation or a sort-merge join algorithm the order of the tuples produced does not depend on the order of the input tuples. The order in which the tupes of a relation are produced is not necessarily the order in which they are given to a consuming operation. This depends on the properties of the mechanism which supports the producer-consumer links.

Fortunately an optimizer does not have to specify explicitly and completely the scheduling of an execution plan. First of all, the execution system may give limited leeway to the optimizer and secondly, some choices might have no or little impact on the cost of the plan. Rather, the optimizer when it has some freedom, chooses a **scheduling strategy** for the execution plan. We define such a strategy as a constraint on the set of all the possible scheduling for the plan. This means that a scheduling strategy specifies a subset of the plan's possible scheduling.

The most usual way to constrain scheduling is to specify which relations will be **materialized** (or blocked) and which ones will be consumed in **pipeline** with their production. A relation is said to be materialized when all its tuples must be produced before the consumption of any of them can start. A relation is consumed in pipeline when it is not materialized, that is when its consumption starts before its production is completed.

Most execution models impose that plans be sliced into sequential phases. In such models, each operation of the execution plan is completely executed during one specific phase, thereby imposing that the execution of an operation cannot overlap with two phases. Consequently, a **slicing strategy** determines non-overlapping fragments (subtrees) of the execution plan, whose operations are executed simultaneously.

3. Slicing hash-join trees

For **linear trees**, i.e. trees in which at least one argument of each join operation is a base relation (as opposed to an intermediate relation), specifying which operations are materialized is enough to determine a slicing strategy. For **bushy** (i.e. non-linear) trees however it is not enough and either the execution system or the optimizer have to further constrain their scheduling. In this section we analyze the problem of choosing a slicing strategy for a tree of hash-joins, and its relationship with the shape (linear, bushy...) of the tree.

[Schn90] adopted a convenient notation, for capturing trees of hash-join algorithms in which the right operand is consumed in pipeline (it is called the **probing** operand) while the left operand is blocked (it is called the **building** operand). The authors then compared the advantages of two extreme cases among the different slicing strategies of linear trees, namely the strategies associated to left-deep and right-deep trees. In a **left-deep** tree (see figure 1), each probing operand is a base relation, so that pipeline is not possible between joins. With these trees, only intra-operation parallelism is possible[2]. In a **right-deep** tree, each materialized operand is a base relation and all the joins can be evaluated in pipeline if enough memory is available.

The size of available main memory is indeed a crucial factor which must be taken into account. Recent relational systems are expected to have enough memory to load one or more relations although probably not enough to load all the relations of a very complex query. [Hong91] experimentally demonstrated that in the context of XPRS the size of main memory, as long as it is above the hash-join threshold [Shap86], has little impact on the choice of the best plan. This is due to the restriction to left-deep trees and is not valid any more when other tree shapes are considered.

To be more precise, in some cases that we discuss below, inter-operation parallelism can be useful to decrease response time. However, to execute several joins in parallel, the hash tables of all the building relations must fit in memory, otherwise the cost of IOs would be prohibitive. Thus, the number of joins that can be executed in pipeline in a right-deep tree, or using independent parallelism in a bushy tree, directly depends on the amount of available memory. Since it is not likely that the hash tables of all the base relations will always fit in main memory, [Sch90] proposed a scheduling strategy called Static Right Deep Scheduling.

This strategy consists in slicing a right-deep tree into phases, such that each resulting fragment is expected to fit in memory, and spooling to disk the temporary results between two phases. In [Zian93] we proposed to avoid spooling the intermediate results to disk which means slicing the RD trees into more phases, possibly diminishing potential parallelism, to save IOs. We showed that this technique can be improved by using **zigzag** trees (see fig 1) which consume less main memory than static right-deep trees. Left-deep, right-deep and zigzag trees are linear trees whose scheduling is much simpler than for bushy trees. However, [Chen92] demonstrated the interest of some bushy trees called segmented right-deep trees. [Shekita93] also considered bushy trees and included memory resources and pipelining in an optimization model for symmetric multiprocessors.

[2]In fact, this is true for the joins but, although this seems to have been overlooked in the litterature, all the scans of a left-deep tree could be run in parallel if enough memory was available. This is quite likely seldom useful however, since then, only one join at a time could be run anyway.

4. Validity of the two-phase hypothesis

Suppose that the two-phase hypothesis is not valid for some query to be run on some system. This means that the best join ordering, say SP0, cannot be parallelized into the best parallel plan, which is in fact a parallelization of another join ordering, say SP1.

How is it possible that the parallel cost of SP0 (i.e. the parallel cost of its best parallelization) is worse than the parallel cost of SP1 while its sequential cost is better ? From the formulas that we gave in our introduction, we can deduce two cases which can lead to a violation of the two-phase hypothesis: either the overhead of parallelism is greater for SP0, or the overhead is not greater but SP0's response time is still greater than SP1's.

In distributed memory it is not very difficult to produce an instance of the first case. In a two-phase optimizer, join ordering does not take into account the communication cost associated to data repartitionning. Then, a cost estimate based of the cardinality of intermediate results could eventually lead to a sub-optimal parallel plan. Consider for example the following query in a system in which communication cost would be expensive:

```
select *
from R1, R3, R3
where R1.A = R2.A and R1.B = R3.B and R2.C = R3.C
```

where R1 has N tuples while R2 and R3 have 2*N tuples. Suppose that the three relations are hashed with the same function on all the processing nodes, and that the declustering attributes are R1.A, R2.A and R3.C. For simplicity, suppose that the optimizer has additional information to deduce that join(R1, R3) has N tuples while join(R1, R2) has 2*N tuples. Then the following join ordering join(join(R1, R3), R2) would be chosen rather than join(join(R1, R2), R3). However, for join(R1,R3), R1 and R3 have to be repartitionned on attribute B while for join(R1, R2), both R1 and R2 are ready to be joined. In both cases another repartitionning is necessary for the second join, and the smaller intermediate result of join(R1, R3) does not compensate the very poor choice of the first join.

In the second case, the worse response time for SP0 is not due to the overhead of parallelism. Then, the reason must be that the parallelization phase was not able to use resources for SP0 as well as for SP1. In shared-memory, these resources cannot be the processors, but it is possible that accessing some relations involves more disks than accessing other relations. This depends on the access methods used (indices, clustering or declustering strategies) and on the form of the selection predicates. In distributed memory, relations might be declustered on different sets of nodes.

It is interesting to note that in the latter case right-deep trees can outperform left-deep trees [Schn90]. More generally, in such a case, inter-operation parallelism can take advantage of more processing nodes and reduce response time. In a system taking only advantage of intra-operation parallelism a dissymmetry in the possibility to use resources would be a serious problem: some operations could not fully use the resources. This can however be overcome in a multi-user environment. For example, in XPRS the fact that some tasks can be IO-bound and other CPU-bound can be solved using inter-operation parallelism and a multi-query scheduler [Hong92]. In the next section we report the results of experiments if which we tested some of the cases we have discussed.

Experimental measurements

Using our parallel optimizer [Zian93], we decided to test the validity of the two-phase hypothesis in shared-memory (SM) and in distributed-memory (DM) architectures. In SM we wanted to model a case in which we expect the two-phase approach to be valid. To do that we did not consider IOs and assumed that all data are memory resident. We also assumed that main memory is infinite, that is large enough to allow the execution of any bushy or right-deep plan. In DM we assumed that relations are hashed on different set of nodes, reproducing a case in which the two-phase hypothesis is not expected to hold. As it was the case in the rest of this paper we did not take indices into account, we only considered hash-based join algorithms, and we ignored data skew.

The cost model we used is an improved version of the model described in [Zian93]. The SM part of our cost model has been validated against actual performance using the standard benchmark AS^{3}AP on an Encore Multimax Multiprocessor with 14 processors [Zait94]. The validation concerned selections and joins as well as the model of pipeline execution of joins which is quite involved. We derived the error rates in predicting the cost of parallel execution plans. Our experiments show that the optimizer predictions are fairly accurate for select-project-join queries. For all experiments, the maximum deviation of the estimated response time from the measured one is 12.15 % by underestimation and 16.83 % by overestimation. The average error rate is 5.85 % by underestimation and 9.03 % by overestimation. The machine parameters for SM and DM which are those of the DBS3 [Zian93] and EDS [EDS] prototypes respectively.

We used two databases each composed of twelve relations, generated automatically following the specifications of the standard Wisconsin Benchmark [Bitt83]. Tuples are 208 bytes long, and relation cardinality is fixed to 10,000 tuples. The size of the intermediate relation of each join is also 10,000 tuples. The two databases only differ in partitionning information, i.e., how base relations are assigned to the processors. In SM, all relations are partitionned on all processors using the first attribute. When it is fixed, the number of processors is 10. In DM, relations are partitionned on 4 disjoint subsets of 5 processors (for a total of 20 processors) each containing three relations. Note that our physical catalogue and our optimizer are able to take into account that the administrator has declustered several relations on the same home. Our optimizer assumes that, in DM, if two relations are not declustered on the same set of nodes these sets are disjoint.

We conducted our experiments using join queries with 4 to 12 relations. We chose a chain form for the query, i.e., each relation, except the first and the last one, is connected to exactly two other relations by one join predicate. We believe that choosing another query form will yield to the same result because the relations are identical. The partitionning attribute is used to join relations. Note that we were only interested in the response time component of the cost function. The optimizer was run on a SUN Sparc-10 workstation with 32 Mbytes of memory.

In all the experiments the x axis represents the number of relations or the number of processors and the y axis represents the estimated response time, in seconds. We used Dynamic Programming (adapted to cope with dependencies discussed in section 3.2) on various search spaces: left-deep, right-deep, and bushy. Note that we modified the optimizer behavior to avoid linear plans when exploring a bushy space, i.e., we forced the optimizer to produce a bushy plan even if a linear one is better.

In a preliminary series of experiments we tested whether restricting the search space to left-deep trees is reasonable in shared-memory (SM) and in distributed-memory (DM). The results confirm the validity of such restriction in SM under our assumptions, but as expected, not in DM. If in SM our optimizer had found counter-examples then the validity of the two-phase approach would have become very unlikely. If no counter-examples were found in DM then our cost model would have become suspicious. After this first set of reassuring experiments we compared the results that two-phase optimization gives, with the results of our (one-phase) parallel optimizer, both in SM and in DM. The results confirmed our predictions.

Figure 2: Response time in SM for several tree formats

Figure 2 shows the response time of execution plans obtained in three search spaces, left-deep, right-deep and bushy, for SM. We see that right-deep execution plans have the worst response time because pipeline parallelism introduces time-sharing between the processes of the different join operations. The non-linear growth of the curve is due to the additional overhead each time a new operation is added to the query. The left-deep plans have the best response time because no time-sharing occurs. When the optimizer explored the space of bushy plans, it only found bushy plans very similar to a left-deep format, that is plans in which only one node had two temporary input relations. This is why the curve of left-deep and bushy plans are quite close.

Figure 3: Response time of in SM for varying number of processors, for a query with 10 relations

Figure 4: Speed-up in SM for a query with 10 relations

Figure 3 shows the response time of the execution plan found for the query with 10 relations in three search spaces, left-deep, right-deep and bushy, when varying the number of processors in SM. The right-deep plans always show the worst response time, and left-deep plans are superior even if bushy plans are close. The response time decreases with the number of processors but the speed-up stabilizes for high number of processors because of the overhead of parallelism (See Figure 4). The speed-up stabilizes for right-deep plans earlier than for bushy or left-deep plans because the overhead associated with time-sharing becomes quickly very important when the number of processors increases. The same principle applies, (to a lesser extent since bushy plans usually have more phases than right-deep ones) to bushy plans compared to left-deep plans.

Figure 5: Response time in DM for several tree formats

Figure 5 shows the response time of execution plans in DM. Left-deep plans only exploit intra-operation parallelism and thus do not take advantage of the fact that some relations are distributed on disjoint sets of processors. Moreover, the overhead due to the communication delay increases response time when transfers of temporary relations occur.

For right-deep plans response time increases very slowly for queries with six or more relations. This result is rather surprising but can be explained: when more relations are joined, more processors are used by the query because the set of processors are disjoint. To be more precise, with four relations the first three ones are on the same home and the fourth one on another home. Thus, two joins are executed on the first home and then the result is transferred to the fourth relation's home and consumed in pipeline. In such a right-deep tree, the first two joins slow down the last one because of time-sharing (assuming a fast interconnection network). When the fifth relation is added to the query, since its home is the same as the fourth relation's home, both homes execute two joins and response time does not increase very much. Only pipeline overhead deteriorates the performance. With six relations, the first home still has two joins to execute but the second one has tree joins now, so that there is a clear increase in response time.

With bushy trees the optimizer has more freedom to balance the number of joins to be executed by each home. When the seventh (or tenth) relation is added, the processors of its home can be used to relieve a three-join home of one join. The curve for bushy space is not complete because Dynamic Programming strategy runs out of memory with a 9-way join query or more.

In the next set of experiments we compared the two-phase approach and the approach of our parallel optimizer. In the latter, choices related to the shape and scheduling of join trees are not assumed to be independent from join ordering. Consequently, the search space is larger in a parallel optimizer and better plans might be found. We simulated the two-phase approach on our parallel optimizer, which implied to set the number of processors to one in the first phase and to modify the physical catalogue to assume that all relations are stored on the same processing element. The second phase mainly consisted in setting new execution details (pipeline/materialized, repartitionning, ...) using the database described above, but keeping the join order found in the first phase.

Figure 6: Response time of bushy trees in DM with parallel and two-phase optimizations

The results showed that the two-phase hypothesis seems valid in SM under the simplifying assumptions we made. We do not show these results since the same plans were found with or without this assumption, for left-deep, right-deep and bushy search spaces. We observed the same result in DM when the search space is restricted to left-deep trees. But when the search space includes bushy trees, the two-phase optimizer always chooses left-deep trees[3] while the parallel optimizer chooses bushy trees. We then checked what happens if the two-phase optimizer is forced to choose a bushy tree (see Figure 6). But of course it was not very helpful since it chose bushy plans that are as close as possible to left-deep plans.

5. Conclusion

In this paper we have shed light on the domain of validity of the two-phase hypothesis. We first clarified the differences between sequential and execution plans and made precise the decisions to be taken by a parallel optimizer. Our analysis

[3]Left-deep trees are always chosen because they provide less overhead due to time sharing and pipeline. In our execution model [Zian93], when there is only one processor, "parallelism" is simulated by time-sharing.

suggests that the two-phase hypothesis is usually not valid in DM. In SM it is true as long as disks are not involved. Coping with CPU-bound and IO-bound tasks is however still possible as shown by [Hong92]. Finally, we made several experiments which confirmed some of the conclusions of our analysis.

References

[DeWi90] DeWitt, DJ and Gray, J "Parallel Database Systems: The Future of Database Processing or a Passing Fad", ACM SIGMOD Record, Vol 19 No 4 (December 1990) pp 104-112

[EDS] EDS Team, "EDS - Collaborating for a High Performance Parralel Relational Database", Esprit Conference, Brussels, November 1990.

[Grae89] Graefe, G and Ward, K "Dynamic Query Evaluation Plans", ACM SIGMOD Int. Conf. on Management of Data, Portland, Oregon, June 1989.

[Grae90] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System", ACM SIGMOD Int. Conf., Atlantic City, New Jersey, May 1990.

[Hong91] W. Hong, M. Stonebraker: "Optimization of Parallelism Query Execution Plans in XPRS", Int. Conf. on Parallel and Distributed Information Systems, Miami, Florida, December 1991.

[Hong92] W. Hong: "Exploiting Inter-Operation Parallelism in XPRS", ACM SIGMOD Int. Conf. on Management of Data, USA, June 1992.

[Lanz91] R.S.G. Lanzelotte, P. Valduriez: "Extending the Search Strategy in a Query Optimizer", Int. Conf. on VLDB, Barcelona, Spain, 1991.

[Lanz93] R.S.G. Lanzelotte, P. Valduriez, M. Zaït: "On the Effectiveness of Optimization SearchStrategies for Parallel Execution Spaces", to appear in Proc. VLDB 1993.

[Schn90] D. A. Shneider, D. J. DeWitt: ``Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines'', Int. Conf. on VLDB, Brisbane, Australia, 1990.

[Shap86] Shapiro L., "Join Processing in Database Systems with Large Main Memories", ACM-TODS, Sept. 1986.

[Ston88] M. Stonebraker et al.: ``The Design of XPRS'', Int. Conf. on VLDB, Los Angeles, September 1988.

[Zian93] M. Ziane, M. Zaït, P. Borla-Salamet, "Parallel Query Processing with Zigzag Trees", VLDB Journal vol. 2 num.3 pp 277-301 July 1993.

[Zait94] M. Zaït and P. Valduriez and D. Florescu, "On the Validation of a Parallel Query Optimizer" in Proc. Bases de Données Avancées, Clermont-Ferrand, France, 1994

[Seli79] P. G. Selinger and M. M. Astrahan and D. D. Chamberlin and R. A. Lorie and T. G. Price, "Access Path Selection in a Relational Database Management System", ACM SIGMOD Int. Conf. on Management of Data, 1979.

[Shasha91] D. Shasha and T. Wang "Optimizing Equijoin Queries in Distributed Databases Where Relations Are Hash Partitioned" ACM Transactions On Database Systems Vol 16 No 2 (1991) pp 279-308

[Chen92] M Chen and M. Lo and P. Yu and H. Young "Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins" in Proc. Int. Conf. on Very Large Data Bases 1992

[Bitt83] D. Bitton and D.J. DeWitt and C. Turbyfill "Benchmarking database systems - a systematic approach" in Proc. Int. Conf. on Very Large Data Bases , 1983

[Shekita93] E. Shekita and K. Tan "Multi-Join Optimization for Symetric Multiprocessors" in Proc. Int. Conf. on Very Large Data Bases, Dublin, Ireland, 1993