Parallel Query Execution in PRISMA/DB. *

Annita N. Wilschut
Peter M. G. Apers
Jan Flokstra

University of Twente
P.O.Box 217, 7500 AE Enschede, the Netherlands

1 Introduction

In the PRISMA-project, a large multi-processor system has been built, is be used to study the performance gains from parallelism. A parallel, main-memory relational database system (PRISMA/DB) runs on this so-called POOMA-machine. This paper studies the possibilities of using parallelism to improve the performance of relational database management systems. Because the equi-join is an important, and time-consuming operation, queries consisting of a number of equi-joins are used to describe how different forms of parallelism can speed up the execution of such queries.

This paper is organized as follows: First, a brief introduction into PRISMA and the DBMS running on it is given. After that, different forms of parallelism are described and the ways in which they can be used is identified. Using this knowledge, the possible parallelism in the execution of join-queries is discussed. Special attention is paid to pipelining. It is shown, that pipelining needs a new hash-join algorithm and that using this algorithm may yield effective parallelism over a pipeline of join operations. Finally, we discuss the implications of using pipelining as a source of parallelism for the optimization of join queries. The paper is concluded with our plans for future work.

2 PRISMA

The PaRallel Inference and Storage MAchine PRISMA is a highly parallel machine for data and knowledge processing.

The PRISMA-machine contains 100 nodes that each contain a data processor, a communication processor and 16 Mbyte of local memory. 50 nodes have a disk and some nodes have an ethernet card that provides an interface with a host computer. Each communication processor connects a node to 4 other nodes. In this way a fast, high-bandwidth network is provided. This hardware can be classified as a *shared-nothing* multi-processor system.

The machine is designed to support a relational *main memory* database management system PRISMA/DB. An extensive introduction to this system can be found in [Kers87] and in [Wils89]. Here, only the features that are important for this paper are summarized.

PRISMA/DB stores the entire database in main memory. Disks are used for backup only. To gain performance and to make storage in main memory feasible, the tuples belonging to one relation are fragmented over more than one node. A fragment is a set of tuples that belong to the same relation and that reside on the same node. A relation does not necessarily use all available nodes. The fragmentation is disjoint and complete, so each tuple belongs to exactly one fragment.

*The work reported in this document was conducted as part of the PRISMA project, a joint effort with Philips Research Eindhoven, partially supported by the Dutch "Stimuleringsprojectteam Informaticaonderzoek (SPIN)".

A fragment has a process associated with it, that executes operations on that fragment. Such a process is called a One-Fragment Manager (OFM). Both base data and intermediate data are managed by OFMs. OFMs for base-fragments are created at system startup; OFMs for intermediate data are created during query execution. The result of an operation is sent to the OFM that needs it for further processing, if it is an intermediate result, or to the user, if it is an end result. Intermediate results may be fragmented. In that case, the output of an operation is distributed over more than one OFM. After finishing a transaction, the OFMs managing intermediate results are disposed of. The base OFMs stay alive waiting for a next transaction that needs their data.

OFMs are allocated to processors when they are created. Process migration is not supported on the POOMA-machine. In this paper, it is assumed that each OFM has its private processor, because we want to understand the behavior of such a system before the more difficult situation in which different OFMs share one processor is tackled. This assumption implies that each base fragment resides on a private processor, because they each have an OFM. In this way, the data allocation problem is solved for this moment. Data allocation is one of our future research issues though.

3 Parallelism

Before discussing the possibilities of using parallelism for query execution, we start with the definition of some useful concepts: Two sorts of parallelism are relevant to this paper [Bora85, Wils89]:

task-spreading A task is decomposed into a number of similar subtasks that are each executed independently on different parts of the data on different processors. The results of the subtasks are eventually combined to form the result. Task-spreading requires a coordinating process that hands out the subtasks and collects the results if necessary. If the subtasks consist of equal amounts of work, the speedup of task-spreading is expected to be proportional to the number of processors involved. This form of parallelism is called (pure) parallelism in [Bora85].

pipelining A task is decomposed into a number of different subtasks that have to be executed consecutively on the same datastream. The subtasks can be assigned to different processors. Every subtask reads its input from its predecessor and sends its output to its successor. Subtasks are activated, when the first data reach them. When the first data reach the last subtask before the first subtask is done, all subtasks work simultaneously until the first subtask is done. Because of this staging in the execution it is hard to predict the performance gain from pipelining.

Orthogonal to this distinction, the sorts of parallelism that are defined above, can be used in different ways for query execution [Wils89, Schn90]:

intra-operator parallelism One operation in a query tree is distributed over more than one processor.

inter-operator parallelism Different operations in one query tree are executed concurrently.

inter-query parallelism Different queries are executed concurrently. This form of parallelism is not considered in this paper.

Intra-operator task-spreading has been studied extensively during the last few years [Brat89, Schn89, Rich87]. This paper concentrates on using *inter-operator pipelining* and *inter-operator task-spreading*, assuming that each individual operation can be implemented efficiently using intra-operator task-spreading. The next section describes how potential inter-operator parallelism in a query can be identified.

4 Possible Parallelism in Join-queries

In order to describe the possibilities of using inter-operator parallelism for query execution, two query representations are discussed in this section. A join-query can be represented in the following ways.

A *join-graph* is a non-procedural representation of a join-query [Ceri84]. The nodes of the graph represent the relations that participate in the join. Two nodes are connected by an edge if a join criterion connects the relations represented by the edge. Edges are labeled with the selectivity of the corresponding join criterion.

A *join-tree* is a procedural representation of a join-query. The leaves of a join-tree represent the relations that participate in the query. Intermediate nodes are operations on their incoming edges; they send their output via the outgoing edge to the next operation. The root of the tree produces the result of the query. In this way, a join-tree describes an execution plan for a query. Like there are many execution plans for a single join-query, one join-graph can be mapped to several different join-trees.

Figure 1 shows a join-query with its join-graph and two join-trees corresponding with this graph.

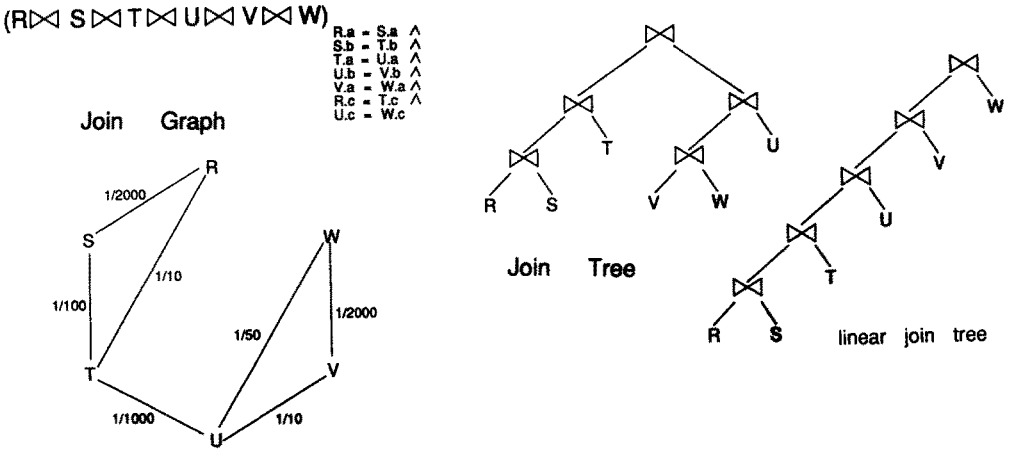

Figure 1: A join-query with its join-graph and two corresponding join-trees.

A join-tree can easily be mapped to the dataflow execution model [Alex88, DeWi88]. In such a model, the data is assumed to flow along the various operation processes, that start processing their input as soon as it is available. The OFMs in PRISMA/DB correspond to operation processes and thus to nodes in a join-tree.

In a join-tree, the potential inter-operation parallelism can easily be identified. Operations that are "next to each other" can be parallelized via task-spreading. Nodes that have a parent-child relationship can possibly execute concurrently via pipelining. Inter-operation task-spreading is used a lot in parallel DBMSs [Ceri84]. In the next section, the possibilities of using inter-operation pipelining to execute join-queries are studied.

5 Pipelining in Join Queries

To find out whether pipelining yields a significant performance gain in join queries, the execution characteristics of the join-tree in figure 2 are considered. This figure shows a join-tree for the four-way join between selections on A, B, C and D. First, the characteristics of the well known hash-join algorithm are described, and then a new version of this algorithm is proposed.

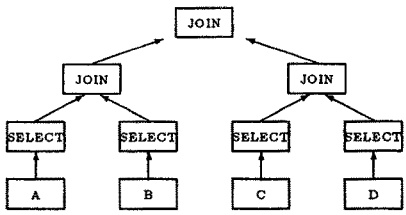

Figure 2: tree representation of $(\sigma A \bowtie \sigma B) \bowtie (\sigma C \bowtie \sigma D)$. The operands are equal in size, the selections have a selectivity of 10%, and the joins operations match one tuple of one operand to exactly one of the other. The operations are executed on a private processor.

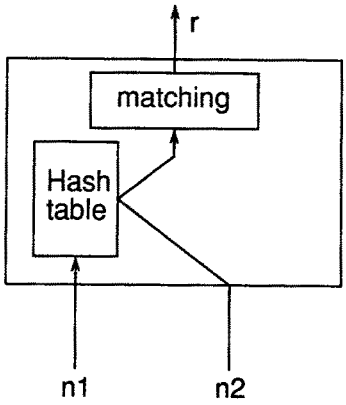

Figure 3: Common main-memory hash-join.

5.1 The common hash-join algorithm

A hash-join algorithm is assumed, because hashing algorithms have proven to perform good compared to other join algorithms [Schn89]. Different hash-join algorithms have been proposed, but they differ mainly in the way in which operands are fetched from disk. The main-memory version of these hash-join algorithms, called *common hash-join* in this text, works as follows (see figure 3):

> In the common hash-join algorithm two phases can be distinguished. First, an in-memory hash-table for one entire operand is built. In the second phase, the tuples of the other operand are hashed and compared to the tuples in the corresponding bucket of the first operand one by one. If a match is found, an output tuple is produced. This algorithm does not need a hash-table for the second operand.

The following can be remarked about this algorithm. Firstly, it is clear that output tuples are only produced during the second phase of the algorithm. Secondly, the algorithm is asymmetric in its operands. This implies that the execution characteristics of $A \bowtie B$ can be very different from the execution characteristics of $B \bowtie A$. Both these properties have important implications for the characteristics of the execution of join queries when the common hash-join algorithm is used. The next paragraph describes the implications of the staging in the algorithm; the asymmetry is discussed further below.

Fig 4 shows the execution characteristics of the join-tree in figure 2, assuming the common hash-join algorithm. The diagrams in this figure plot the processor utilization of an operation against the time. The two phases in the join operation can easily be distinguished in the join-diagrams. It is clear, that the topmost join operation in the tree can only start building its hash-table during

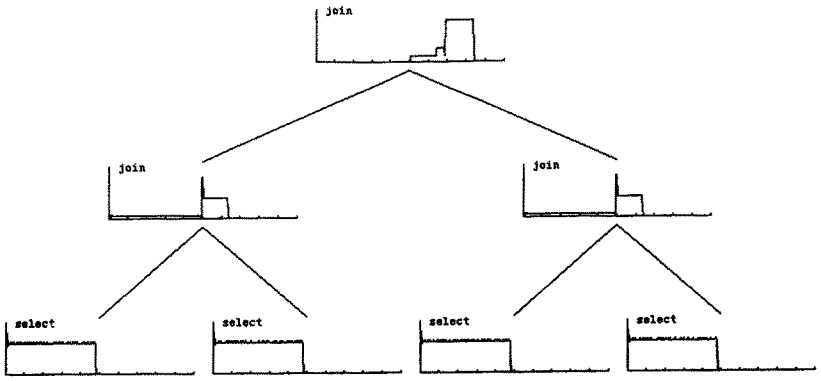

Figure 4: Execution characteristics of the common main-memory hash-join algorithm.

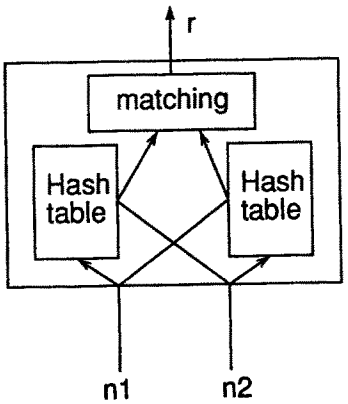

Figure 5: Pipelining main-memory hash-join.

the second phase of the other two join operations. The second phase of the topmost join operation is executed after the other two join operations have finished executing, and so this phase cannot work concurrently with the other two join-operations. So, in a pipeline of join operations that are implemented via this join algorithm, the first phase of a parent join can be executed concurrently with the second phase of the child join operation. Therefore, if a pipeline consists of more than two join operations, at any moment at most two operations can execute concurrently. This means that the staging in the common hash-join algorithm reduces the effective parallelism from pipelining.

5.2 A pipelining hash-join algorithm

To increase the amount of effective parallelism in the execution of the join-tree in figure 2, a new main-memory hash-join algorithm, called *pipelining hash-join* is proposed [Wils90] (see figure 5):

> The pipelining hash-join consists of only one phase in which a hash-table for both operands is built. When a tuple arrives of one of the operands, it is hashed and compared to the tuples in the corresponding bucket of the other operand that have already arrived. If a match is found an output-tuple is formed. Finally, independant of the match, the input-tuple is inserted in its own hash-table. As soon as one entire operand has reached the join process, the tuples of the other operand do not need to be inserted in their own hash-table, because this hash-table is not used in the rest of the join process anymore.

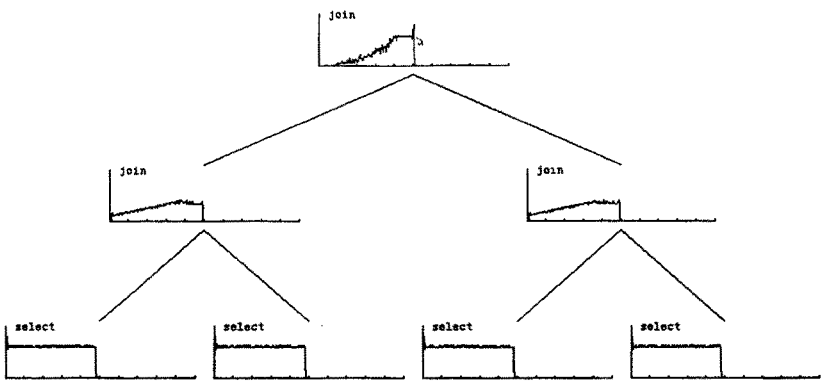

Figure 6: Execution characteristics of the pipelining main-memory hash-join algorithm.

The following properties of this algorithm are important: Firstly, this algorithm already produces its first output tuple as soon as two matching tuples have reached the join operation. Secondly, the algorithm is symmetric in its operands. Finally, if one entire operand reaches the join-process before the first tuple of the other operand is available, this algorithm degenerates to the common hash-join algorithm.

The next paragraph describes how the first property of this algorithm influences the effective parallelism from pipelining. The next subsection is on the (a)symmetry of the hash-join algorithms that are presented in this paper.

Figure 6 shows the execution characteristics of the join-tree in figure 2 using the pipelining hash-join. From this figure it is clear, that all three join operations work concurrently, resulting in a shorter response time for the execution of the complete query. This increased concurrency (with respect to the execution of this join-query using the common hash-join algorithm) is possible, because the result tuples are produced early on the join process. Using the pipelining hash-join algorithm, it is feasible to build join pipelines that consist of several join operations that can all work concurrently. So, using the pipelining hash-join algorithm, both task-spreading and pipelining yield effective inter-operator parallelism.

5.3 The advantage of a symmetric join algorithm

This subsection is a deviation of the main topic of this paper. In the next section, the discussion on parallel query execution is continued with the implications of the pipelining hash-join algorithm for query execution strategies. Here, the advantages of the symmetry of the pipelining hash-join over the asymmetry of the common hash-join are described. It should be realized, that the join operation is a symmetric one in theory; the asymmetry of the common hash-join is merely a consequence of the way in which the tuples of the operands are matched.

The asymmetry of the common hash-join has disadvantages for the determination of a suitable plan for a join-query. Usage of an asymmetric algorithm requires that the decision which operand has to be the left one and which one the right, has to be made thoughtfully. Taking the wrong decision can result in bad execution chararteristics: If for a certain join operation, a common hash-join is planned in which a hash-table is built for the left operand, and the right operand is available earlier than the left one, than the join process sits waiting for the left operand, not doing anything with the tuples of the right operand, that are available for processing. If the pipelining hash-join is used instead, the algorithm itself adapts to the availability of tuples of the operands, because the tuples are processed in the order in which they are available. This adaptation of the algorithm to the availability of the operands makes the task of determination of a suitable query execution strategy

easier.

In a paper on query execution strategies, DeWitt and Schneider [Schn90] compare the effective inter-operator parallelism during the execution of linear left-deep and linear right-deep join trees. They conclude that right-deep linear join trees are well suited to process multiway joins. The difference in the execution characteristics of these two strategies is caused by the asymmetry in the hash-join algorithm they use. In a main-memory environment, usage of the symmetric pipelining hash-join algorithm probably yields the same effective parallelism, regardless of the shape of the query tree. In the next section, the implications of the fact that arbitrary shaped query trees can be used for parallel query execution, are discussed.

6 Query Execution Strategies

In this section, the impact of using the pipelining hash-join algorithm on the design of a query execution strategy is considered.

Choosing an execution plan for a query implies that the following two decisions (among others) have to be taken:

- An algorithm has to be chosen for each individual join operation. This choice implies determination of the degree of intra-operation parallelism for each operation.
- A join-tree that corresponds to the join-graph for the query has to be selected from the numerous possible join-trees.

These choices can be fixed (e.g. some systems always use a hash-join algorithm that is distributed over all available processors), or they are made by a query optimizer. To select a join-tree, usually a part of the vast space of join-trees that corresponds to a join-graph is traversed more or less exhaustively searching for the tree that has the best value for some goal-function. The goal-function calculates some execution characteristic of the join-tree using estimates of the costs of the individual operations. Of course, the cost estimates are influenced by the algorithms that are assumed for individual operations.

Many papers on query optimization discuss the process of traversing the space of join-trees and cost functions that are used to estimate the costs of operations and the size of results. In this paper, we assume suitable cost functions to be available and we only try to identify the characteristics of the join-tree that has to be chosen, leaving the way to find that tree for further research.

Now, we will discuss two query execution strategies that are known from the literature, and then an alternative strategy that takes inter-operation pipelining into account, is proposed.

6.1 GAMMA.

GAMMA [DeWi86] uses the approach that is known from System-R [Seli79]. It can be characterized as follows:

- A (non-pipelining) hash-join algorithm is used in which each operation is declustered over all 8 available processors.
- The *linear* tree that has minimal accumulated estimated processing costs is chosen as execution plan for the query.

This strategy has a fixed choice for the join algorithm and for the degree of intra-operator parallelism that is used. The query optimizer only considers linear join trees. Also, this strategy does not use any inter-operator parallelism: each operation occupies all available processors. Therefore, using inter-operator parallelism will not yield any performance gain.

6.2 The Bodorik approach

Bodorik et al.[Bodo88] propose an optimization strategy that chooses a general (not necessarily linear) join-tree in which the accumulated processing time along the longest path is minimal with respect to other join-trees. It is clear, that such an algorithm tends to select wide query-trees that have many possibilities for task-spreading, which is the main source of parallelism in the execution model that is used in this paper. The Bodorik query execution strategy is characterized as follows:

- A (non-pipelining) hash-join algorithm is used in which the degree of parallelism in each individual operation depends only on the fragmentation of the join operands. Before each join operation, one operand is reconstructed and broadcast to the fragments of the other operand.
- The optimization algorithm selects the join-tree in which the total execution time of the operations on the longest path is minimal.

In this strategy, the join algorithm is fixed, and the degree of parallelism in each operation is taken according to the fragmentation of the join operands. The strategy uses both intra-operator and inter-operator task-spreading.

6.3 The PRISMA approach

In PRISMA yet another strategy is used. The execution strategy that is proposed here assumes usage of the pipelining hash-join algorithm. The idea behind it is the following:

- First, the total amount of work that has to be done to evaluate the query is minimized.
- Then, we try to distribute that minimal amount of work equally over the available processors.

If all processes that result from this strategy really execute concurrently, it is clear that this strategy is likely to yield a good response time to the query.

In the previous section, it was stated that both inter-operator task-spreading and inter-operator pipelining yield concurrent join processes, if the pipelining join algorithm is used. So, the idea that is described above can be applied in the following way.

- First, the join-tree is chosen that has the minimal accumulated processing time over all joins that have to be executed.
- Now, the work in the chosen join-tree has to be distributed equally over the available processors. This is achieved by assigning more processors to expensive operations: The available processors are assigned to join operations proportionaly to the costs of those join operation.

So, the PRISMA approach can be characterized as follows:

- A main-memory pipelining hash-join algorithm is assumed. The degree of intra-operation parallelism in each join is determined by the estimated costs for each operation.
- The minimal total estimated processing costs general tree is chosen.

Comparison of the three optimization strategies that are described above leads to the following observations. Firstly, GAMMA only exploits intra-operator parallelism, Bodorik et al. use intra-operator task-spreading and inter-operator task-spreading, and PRISMA exploits intra-operator task-spreading, inter-operator task-spreading and inter-operator pipelining. Secondly, GAMMA and Bodorik set the degree of intra-operator task-spreading heuristically and PRISMA chooses this degree during the optimization process. Finally, different sorts of query trees are chosen. Experiments will have to show which strategy works best.

Of course, the PRISMA optimization strategy has some problems as well. It is clear that two pipelined processes do not yield full concurrency. The effect of the intrinsic delay over a pipeline and the tuning of the speeds in which tuples are generated and consumed need further study.

7 Summary and plans for future work

In this paper, the possibilities for parallel query execution in PRISMA/DB were reviewed. Join queries were considered throughout the paper. It was shown that inter-operator pipelining may yield a source of parallelism in query execution. Exploitation of this form of parallelism affects the characteristics of query execution. An adjusted query execution strategy was proposed.

The material presented in this paper is still is the stage of ideas that need experimental justification. Also, many details of new the algorithms that were proposed need further research. Experiments are carried out in two ways. Firstly, PRISMA/DB is an excellent experimentation platform for the ideas that were presented in this paper. We plan to compare the characteristics of the execution of different query trees for one query on PRISMA/DB. Secondly, a simulator for query execution was developed with which the influence of changes in algorithms can easily be studied. (Figure 4 and 6 were produced by this simulator.) Both PRISMA/DB and the simulator will be used to adjust and validate the algorithms that are presented in this paper.

After that more general queries than just join queries will be considered. Also, process and data allocation will be studied. For that purpose the possibility that more than one data fragment or more than one operation process share one processor is taken into account.

References

[Alex88] W. Alexander and G. Copeland, *Process and Dataflow Control in Distributed Data-Intensive Systems*, Proceedings of the 1988 SIGMOD conference, Chicago, USA, June 1988.

[Bodo88] P. Bododrik, J. S. Riordon, *Heuristic Algorithms for Distributed Query Processing*, Proceedings of the first International Symposium on Databases in Parallel and Distributed Systems, Austin, USA, December 1988.

[Bora85] H. Boral and S. Redfield, *Database Machine Morphology*, Proceedings of the 11th conference on Very Large Databases, Stockholm, Sweden, August 1985.

[Brat89] K. Bratbergsengen and T. Gjelsvik, *The Development of the CROSS8 and HC16-186 Parallel Database Computers*, Proceedings of the 6th International Workshop on Database Machines, Deauville, France, June 1989.

[Ceri84] S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems*, McGraw-Hill, 1984.

[DeWi86] D.J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, M. Muralikrishna, *GAMMA - A High Performance Dataflow Database Machine*, Proceedings of the 12th conference on Very Large Databases, Kyoto, Japan, August 1986.

[DeWi88] D.J. DeWitt, S. Ghanderarizadeh and D.Schneider, *A performance analysis of the Gamma Database Machine*, Proceedings of the 1988 SIGMOD conference, Chicago, USA, June 1988.

[Kers87] M.L. Kersten, P.M.G. Apers, M.A.W. Houtsma, H.J.A. van Kuijk, R.L.W. van de Weg, *A Distributed, Main Memory Database Machine*, Proceedings of the 5th International Workshop on Database Machines, Karuizawa, Japan, October 1987.

[Rich87] J. P. Richardson, H. Lu and K. Mikkilineni, *Design and Evaluation of Parallel Pipelined Join Algorithms*, Proceedings of the 1987 SIGMOD conference, San Francisco, USA, June 1987.

[Seli79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, *Access Path Selection in a Relational Database Management System*, Proceedings of the 1979 SIGMOD conference, Boston, USA, 1987.

[Schn89] D.A. Schneider and D.J. DeWitt, *A performance Evaluation of Four Join Algorithms in a Shared-Nothing Multiprocessor Environment*, Proceedings of the 1989 SIGMOD conference, Portland, USA, June 1989.

[Schn90] D. A. Schneider and D. J. DeWitt, *Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines*, Proceedings of the 16th conference on Very Large Databases, Brisbane, Australia, August 1990.

[Ston86] M. Stonebraker, *The case for shared nothing*, Database Engineering 9.1, 1986.

[Wils89] A.N. Wilschut, P.W.P.J. Grefen, P.M.G.Apers, M.L.Kersten, *Implementing PRISMA/DB in an OOPL*, Proceedings of the 6th International Workshop on Database Machines, Deauville, France, June 1989.

[Wils90] A. N. Wilschut, P. M. G. Apers, *Pipelining in Query Execution*, Proceedings of the PARBASE-90 conference, Miami, USA, March 1990.