Parallel evaluation of multi-join queries

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

University of Twente
P.O.Box 217, 7500 AE Enschede, the Netherlands
Phone: +3153-894190
{annita, flokstra, apers}@cs.utwente.nl

Abstract

A number of execution strategies for parallel evaluation of multi-join queries have been proposed in the literature; their performance was evaluated by simulation. In this paper we give a comparative performance evaluation of four execution strategies by implementing all of them on the same parallel database system, PRISMA/DB. Experiments have been done up to 80 processors. The basic strategy is to first determine an execution schedule with minimum total cost and then parallelize this schedule with one of the four execution strategies. These strategies, coming from the literature, are named: *Sequential Parallel*, *Synchronous Execution*, *Segmented Right-Deep*, and *Full Parallel*. Based on the experiments clear guidelines are given when to use which strategy.

1 Introduction

For years now, research has been done on the design, implementation, and performance of parallel DBMSs. Teradata [CaK92], Bubba [BAC90], HC186-16 [BrG89], GAMMA [DGS90], and XPRS [SKP88] are examples of systems that actually were implemented, and many papers were written on their performance. The performance evaluation of these systems is mainly limited to simple queries that involve no more than one or two join operations.

Recent developments in the direction of support of non-standard applications, the use of complex data models, and the availability of high-level interfaces tend to generate complex queries that may contain larger numbers of joins between relations. Consequently, the development of execution strategies for the parallel evaluation of multi-join queries has drawn the attention of the scientific community. A number of strategies was proposed [CLY92,CYW92, HoS91,HCY94,ScD90] and their performance was evaluated via simulation. However, no comparative experimental performance evaluation is available. This paper describes the proposed strategies in a common framework. Four strategies are implemented on PRISMA/DB and a comparative performance evaluation is done. The results yield clear guidelines for the choice of a strategy.

SIGMOD ' 95,San Jose , CA USA

1.1 Implementation platform

PRISMA/DB was used to do the experiments. PRISMA/DB is full-fledged parallel, relational DBMS [ABF92]. A fully functional prototype is running on a 100-node multi-processor machine. PRISMA/DB is used for research in various directions [Gre92,HWF93,Wil93,WiA91,WFA92]. Here the potential of the system —parallelism up to a large number of processors and the possibility to implement a wide variety of parallel execution strategies— is used to study the parallelization of multi-join queries. PRISMA/DB is a main-memory DBMS and therefore the experiments described in this paper refer to a main-memory context. The concluding section of this paper discusses the applicability of the results of our work for disk-based systems.

1.2 Optimization and parallelization of multi-join queries

System R [SAC79] is the pioneer in the area of optimization of multi-join queries in a centralized environment. In System R, join trees are restricted to linear trees, so that available access structures for the inner join operand can optimally be exploited. System R chooses the cheapest (in the sense of minimal total costs) linear tree that does not contain cartesian products.

Subsequently, it is remarked in [KBZ86] that the restriction to linear trees may not be a good choice for parallel systems. However, the space of possible join trees is very large if restriction to linear trees is dropped [LVZ93]. In [LST91, SwG88] partially heuristic algorithms are proposed that aim at limiting the time spent on searching the space of possible query trees for the cheapest one. [SHV92] proposes to parallelize this search. In these papers, the cost formula used evaluates the total costs of a query tree, not taking the influence of parallelism into account.

Obviously, when optimizing the response time of a complex query, it is not sufficient to optimize towards minimal total costs. Rather, the exploitation of parallelism has to be taken into account as well. However, the search

space that results if all possible trees and all possible parallelizations for these trees are taken into account is gigantic. To overcome these problems, [HoS91] proposes a two-phase optimization strategy for multi-join queries. The first phase chooses the tree that has the lowest total execution costs and the second phase finds a suitable parallelization for this tree. Although not all researchers agree on this assumption [SrE93], this paper will adopt it for the following reasons. First, it does not seem reasonable to assume that parallelism will to a large extent compensate for an increased total amount of work. Second, the schedule with minimal total costs is likely to have small intermediate results, so that the transmission costs in the parallel execution of this schedule will be low as well. Third, two-phase optimization seems a reasonable way to cut down on the optimization time. Lastly, missing the very best execution plan is not a big problem as long as you can assure that you will not come up with a very bad one [KBZ86]. The first phase of the two-phase optimization can easily be handled by standard query optimization. The second phase: finding a suitable parallelization for a given join tree is the subject of this paper.

1.3 Organization of paper

This paper is organized as follows. Section 2 shortly introduces PRISMA/DB, it shows how the different strategies for the execution of multijoins can be implemented on PRISMA/DB, and it discusses some results from earlier research in the context of PRISMA/DB that are used to explain the results of this paper. Section 3 describes four execution strategies for multi-join queries and their trade-offs in detail. Section 4 describes a comparative performance evaluation and Section 5 summarizes and discusses the results of this paper.

2 PRISMA/DB

PRISMA/DB has extensively been used for research in the area of parallel query processing [Wil93,ApW94]. Our previous research followed two lines. First, the system was used to experiment with large-scale intra-operation parallelism for single operation queries [WFA92]. Second, a theoretical study of the behavior of pure inter-operation parallelism in multi-join queries was done [WiA93]. The work presented in this paper combines those two lines of research: we study the use of both *inter-* and *intra-*join parallelism for the execution of multi-join queries via experimentation. This section describes the system and its hardware, the PRISMA/DB query execution engine, and those results from previous research that used to explain the results of the work presented in this paper.

2.1 The system

PRISMA/DB is a full-fledged parallel, main-memory relational DBMS, designed and implemented in the Netherlands. A goal of the PRISMA project was to provide flexibility in architecture and query execution strategy, to enable experiments with the functionality and performance of the system. This flexibility is used here to implement various strategies for the parallel evaluation of multi-join queries and to evaluate their performance. PRISMA/DB currently run on a 100-node shared-nothing multi-processor. Each node consists of a 68020 processor with 16 Mbytes of memory, a disk, and a communication processor. A full description of design, architecture, and implementation of PRISMA/DB can be found in [Ame91,ABF92].

2.2 Flexible query execution in PRISMA/DB

The query execution engine of PRISMA/DB consists (for each query) of a single scheduler and multiple operation processes on each processor. The operation processes can access data fragments that are stored in the main memory of their own processor directly and execute relational operations on them. Results can be stored in the local memory or split and sent to other processors for further processing. A pool of operation processes is kept alive in the system; a scheduler can claim free operation processes for the execution of a query. The scheduler initializes the participating operation processes with the relational operation to be executed. The coordination between the operation processes is done by the operation processes themselves. In this way, the coordination is parallelized.

An eXtended Relational Algebra (XRA) ([GWF91]) is used as internal representation of queries. This language consists of the normal relational operations extended with some primitives for grouping and for recursive query processing. Also, the language allows the expression of a wide range of parallel execution plans for a query. Each relational operation can be executed by an arbitrary number of processors, and the result of an operation can be distributed efficiently over an arbitrary number of destinations. Also, operations can be allocated explicitly to processors. In this way, *intra-operator parallelism* with an arbitrary degree is achieved. Allocating two independent join operations to disjoint sets of processors results in inter-operator parallelism. *Inter-operator pipelining* can be implemented via allocation of pipelined joins to disjoint sets of processors. This flexibility yields the possibilities to implement a wide variety of strategies for multi-join queries.

2.3 Results from previous research

2.3.1 Parallel execution of single operator queries

[WFA92] studies the use of intra-operator parallelism for main-memory database systems. In that study, it is concluded that observed linear speedup for small numbers of processors cannot always be extrapolated to larger numbers of processors. This is caused by the fact that the overhead from starting on operations on processors —this overhead increases with increasing degree of parallelism— dominates the actual processing time —which decreases with increasing degree of parallelism— for a large degree of parallelism.

The optimal number of processors to be used appears to be proportional to the square root of the size of the operands. As a consequence, larger problems allow a larger degree of parallelism. Also, it is concluded that the optimal number of processors for the parallel execution of an operation is smaller for a main-memory system than for a disk-based system.

2.3.2 The Pipelining hash-join algorithm

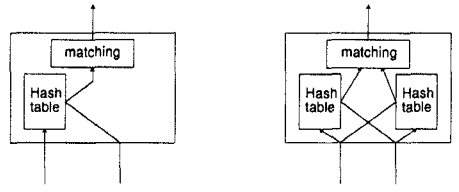

Figure 1: Simple hash-join and Pipelining hash-join algorithm in a main-memory system

In [WiA91,WiA93] it is shown how special main-memory algorithms can be used that enhance the effective parallelism from pipelining. These pipelining algorithms aim at producing output as early as possible, so that a consumer of the result can start its operation. In particular, [WiA91, WiA90] proposes a pipelining Hash-Join algorithm. As opposed to the well-known two-phase, build-probe hash-join [ScD89,WiA91] (this algorithm is called simple hash-join in this paper), this symmetric algorithm builds a hash-table for both operands (See Figure 1). The join process consists of only one phase. As a tuple comes in, it is first hashed and used to probe that part of the hash table of the other operand that has already been constructed. If a match is found, a result tuple is formed and sent to the consumer operation. Finally, the tuple is inserted in the hash table of its own operand. Compared to the simple hash-join, the pipelining algorithm can produce result tuples earlier during the join process at the cost of using more memory to store a second hash-table. Using this algorithm, pipelining along both operands of the join is possible.

2.3.3 Linear and bushy trees for multijoin queries

[WiA93,WiG93] present an analytical study of the use of inter-operation parallelism for linear and bushy join trees. For bushy trees the pipelining hash-join algorithm presented above is used to allow pipelining along both operands. It was shown that each step in a *linear* pipeline (so a join that has one base-relation operand and one intermediate result as operand) causes a constant delay. A step in a *bushy* pipeline (so a join that has two intermediate results as operands), however, causes a delay that is *proportional to the size of the operands.* As a consequence, when the join operands are small, a bushy tree works better, and for larger operands linear trees work better. It depends on the number of join operations in the tree and on the sizes of the join operands whether the performance of a linear tree or a bushy tree is better.

3 Parallel execution strategies for multi-joins

The parallel execution strategies for multi-join queries that are dealt with in this paper all use known parallel algorithms to evaluate the constituent binary join operations. The difference between the various strategies lies in the way in which binary joins are allocated to processors. A lot of work was done on the use of intra-operator parallelism for the evaluation of binary join operations. It is generally agreed on that the parallel hash-join is the algorithm of choice [ScD89]. Two version of this algorithm are considered here: the simple hash-join and the pipelining hash-join (see Section 2.3.2).

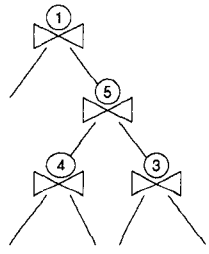

Figure 2: The 5-way join tree that is used to explain the parallel multi-join strategies in this paper.

A parallel execution strategy for a multi-join query uses a parallel hash-join algorithm for the constituent binary joins. Apart from this *intra-operator parallelism*, also inter-operator pipelining or parallelism may or may not be used. The four strategies that are regarded here differ in the way in which *inter-operator parallelism* and *intra-operator parallelism* are used. Note, that we concentrate on adding inter-operator parallelism. This means that the available processors may have to be distributed over the operations in the join-tree. We do not allow a single processor to work concurrently on different join operations.

In the following, each of the strategies is described in detail. The 5-way join tree in Figure 2 is used as an example. The constituent joins in this tree are labeled with a number, which indicates the relative amounts of work in the join operations. So, the second join operation from the top needs five times the computation time of the top join operation. Note that in the processor utilization diagrams in the subsections to come, just for the sake of simplicity, these numbers are also used as identification of the join operations.

3.1 Sequential Parallel Execution (SP)

The sequential parallel execution strategy is the simplest way to evaluate a multi-join in parallel. This strategy does not use any inter-operator parallelism. The constituent joins are executed sequentially in parallel, using all available processors for each join operation. This strategy does not

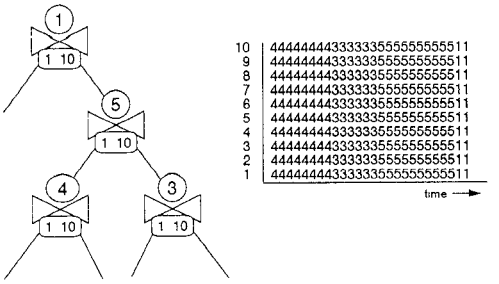

Figure 3: Sequential parallel evaluation of the example join tree.

require pipelining between join operations, so the simple hash-join algorithm can be used. Figure 3 shows the processor allocation (indicated below each join-sign) and idealized processor utilization for this query on a 10-processor system. In the processor utilization diagram, the x-axis represents time and the y-axis represents the 10 processors. Each processor is represented by a line in the diagram and the individual binary joins are indicated by the label they have in the join tree. The diagram shows which processor is working on which join at a certain time. The processor utilization diagram is idealized in the sense that overhead incurred by the parallel execution is not taken into account. From the figure we can see, that the processors first work together on the join labeled with 4, then they work on the join labeled with 3 etc. This strategy does not need a cost function to estimate the costs of the join operation. Also, the idealized load balancing is perfect.

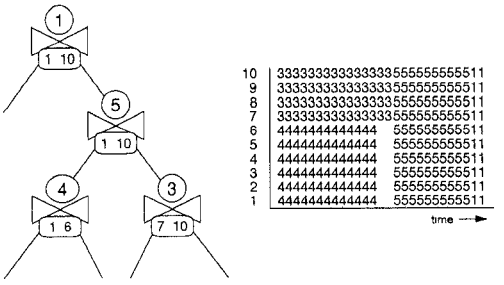

Figure 4: Synchronous evaluation of the example join tree.

3.2 Synchronous Execution (SE)

This strategy uses inter-operator parallelism apart from intra-operator parallelism. The strategy was proposed in [CYW92]. The idea is to execute independent subtrees in the join tree independently in parallel. A join operation is started only after its operands are ready. The only inter-operation parallelism that is used in a join tree is the parallelism between independent subtrees of a bushy tree. An algorithm is proposed in [CYW92] that aims at equal processing time for both operands to be ready for joining. This is done by allocating a number of processors to a subtree that produces an operand, that is proportional to the total amount of work in the subtree. In this way, operands are supposed to be available at the same time so that no processors have to wait. This strategy does not require pipelining between join operations, so the simple hash-join algorithm is used.

Figure 4 shows a possible processor allocation for the example query and the idealized processor utilization for this query. First, the available processors are distributed over joins 3 and 4, and then the other joins are executed sequentially on the entire system. The allocation algorithm needs a cost-function to estimate the processing costs for subtrees in the join tree. The processor utilization diagram shows that even the idealized processor utilization diagram does not achieve perfect load balancing due to discretization errors (see Section 3.5) in the allocation of differently sized loads to a small number of processors.

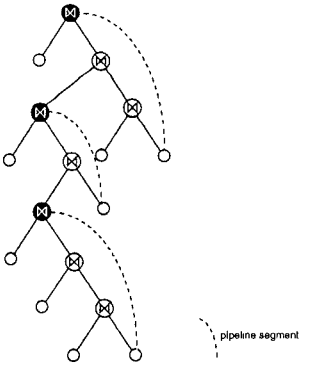

Figure 5: Bushy tree with its right-deep segments.

3.3 Segmented Right-Deep execution (RD)

In contrary to SE, segmented right-deep execution uses inter-operator pipelining in addition to intra-operator parallelism. This strategy is proposed in [CLY92], a paper which was inspired by [ScD90].

Schneider [Sch90,ScD90] describes the differences in possible parallelism between left-deep and right-deep linear join trees[1], when the simple hash-join is used for the individual join operations. In a right-deep tree the build-phases of all join operations can be executed in parallel and after that probe-phases can be executed using extensive pipelining. Left-deep trees on the other hand only allow parallel execution of the probe phase of one join-operation and the build-phase of the next. It is concluded in this study that, due to the possibilities of extensive exploitation of pipelining right-deep trees perform better than left-deep trees.

The results of Schneider are extended in [CLY92] to bushy trees. That paper proposes to see a bushy tree as a

[1] In this terminology the inner join-operand, which is used to build a hash-table, is called the left operand, and the outer join-operand, which is used to probe the hash-table is called the right operand.

segmented right-deep tree, which is a bushy tree that consists of right-deep segments (see Figure 5). The right-deep segments can be evaluated using inter-operation parallelism as proposed in [Sch90,ScD90]. Each operation in a segment is assigned a number of processors that is proportional to the estimated amount of work in the join operation. Segments that have a producer-consumer relationship are evaluated sequentially. Independent segments, however, may be evaluated in parallel, using disjoint subsets of the available processors. In this approach, a left-deep tree is a bushy tree consisting of many small right-deep segments.

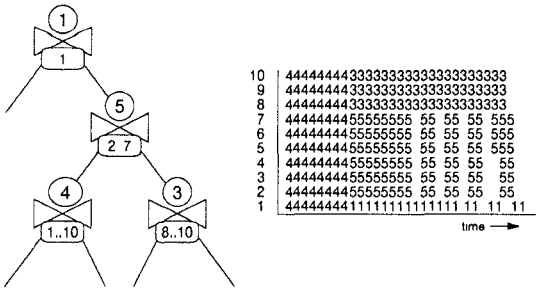

Figure 6: Segmented Right-Deep evaluation of the example join tree.

Figure 6 shows a possible processor allocation for the example query and the idealized processor utilization for the chosen allocation. This strategy first uses all available processors to process the right-deep subtree that consists just of the join labeled with 4. Subsequently, the available processors are distributed over the other join operations, which also form a right-deep subtree. This last subtree is executed in a pipelined fashion. Each of the join operations starts immediately hashing its left operand. However, during the probe-phase the join labeled with 3 (which has relatively few processors) cannot saturate the joins that are higher up in the pipeline so those operations cannot fully utilize their processor during the probe phase. This effect is indicated in the diagram by holes in the execution lines.

Again, this strategy needs a cost function to estimate the amount of work in each join operation. This strategy also does not yield perfect load balancing due to discretization errors in the allocation of work to a small number of processors and due to delays over the pipeline (tuples cannot be processed by a consumer before they are generated by the producer).

3.4 Full Parallel Execution (FP)

This strategy adds both inter-operator pipelining and inter-operator parallelism to intra-operator parallelism in the individual join-operations. The strategy was proposed in [WiA91,WAF91]. The idea behind this strategy is to allocate each join-operation to a private (set of) processors, so that all join-operations in the schedule are executed in parallel. Depending on the shape of the query tree, pipelining and independent parallelism are exploited. The strategy uses the

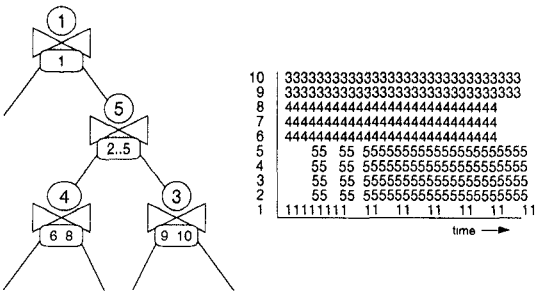

Figure 7: Full parallel evaluation of the example join tree.

pipelining hash-join algorithm (see Section 2.3.2). Because, this algorithm can exploit pipelining along both the right and the left operand, all individual join-operations can be executed in parallel. The available processors are distributed over all join-operations proportionally to the amount of work in each operation. Each join-operation starts working as soon as input is available.

Figure 7 shows a possible processor allocation for the example query and the idealized processor utilization for the chosen allocation. The bottom two join operations start immediately on the processors allocated to them, as their operands are available as base-relations. The join operation labeled with 5 has to wait some time until its operands start producing output (see Section 3.5). The top join operation may start immediately hashing its left-operand. However, it has to wait for its right operand to become available, and therefore its processor is not fully utilized later during the join operation.

Again, this strategy depends on a cost function to estimate the amount of work in each join operation. It is clear that this strategy does not offer perfect load balancing either.

3.5 Tradeoffs

There are a number of barriers that prevent performance gain from parallelism. A general discussion of this issue can be found in [DeG92]. These barriers affect the execution strategies introduced above in a different way, resulting in a number of tradeoffs.

startup The startup time is the time needed to start join operations on processors. If many operations have to be started, this startup time may dominate the actual computation time (see Section 2.3.1) [2]. The SP strategy uses many operation processes: the number of operation processes used is equal to the product of the number of operations in the join tree and the number of processors used. The FP strategy only uses one operation process per processor. So, the startup overhead is large for SP and small for FP, and SE and RD are in the middle.

[2] In contrast to [WFA92], PRISMA/DB now keeps operation processes running to minimize this startup overhead

coordination Parallel execution of multi-join queries needs redistribution of operands between subsequent join-operations. There is always some *coordination overhead* due to the synchronization of this tuple transport. In PRISMA/DB, for each tuple stream the sender and receiver have to shake hands before the tuple transport can start. The number of tuple streams increases dramatically with the degree of parallelism of the sender and the receiver: if the sender consists of n operation processes and the receiver consists of m operation processes, there will be $n \times m$ tuple streams (see Section 4.3). This overhead starts to count for large numbers of processors. Because SP uses the most processors per operation, SP suffers most from coordination overhead. FP suffers least, and again, SE and RD are in the middle.

discretization error From the utilization diagrams that are presented above, it is clear that only SP achieves perfect load balancing in the ideal case. This is caused by the fact that all processors get exactly the same amount of work (assuming non-skewed data partitioning). The other strategies will never achieve perfect load balancing due to *discretization errors* in the distribution of operations over processors. A simple example can make the point clear. If you have 4 pieces of candy to distribute over 3 kids, one of them will get 2 pieces and the other two each get 1 (assuming we cannot chop the candy to chunks). The uneven distribution of candy over kids is caused by the fact that there are discrete numbers of kids and candies. The strategies presented in this papers distribute processors over operations, and because these are both discrete entities, in general the distribution will not be fair due to discretization errors in the distribution. This leads to load imbalance or *skew*. Obviously, this error decreases with increasing ratio between number processors and number of operations. So, if the number of processors to distribute is large or the number of operations is small, the discretization error will be small. From this it follows that SP does not suffer from the discretization error (SP does not distribute processors over different operations), RD and SE, suffer moderately from this error (as only a subset of the operations share the system at the same time), and FP suffers most, because the available processors are distributed over all operations.

delay over pipelines Finally, RD and FP exploit pipelined parallelism. This form of parallelism incurs a *delay over the pipeline*: an operation process at the top of the pipeline has to wait for the tuples to arrive. The size of this delay depends on the shape of the query tree, the number of join in the pipeline and on the size of the join operands as discussed in Section 2.3.3.

Obviously, each of these four factors affects the execution strategies studied in a different way. Also, it is expected that the extent to which a strategy is affected by each of the factors depends on the shape of the query tree that is parallelized. For example, RD is expected to work fine for right-oriented trees, but not so well for e.g. a left-linear tree. Similarly, SE is expected to work better for bushy trees than for trees that are (almost) linear. SP, on the other hand, is not expected to be very sensitive to the shape of the query tree. Experiments are used to find out how these tradeoffs work out in reality.

4 Performance evaluation

As stated in the introduction of this paper, we study the second phase of a two-phase optimization/parallelization strategy. The first phase, finds the join tree with minimal total costs for a given multi-join and the second phase generates a parallel execution strategy for this plan. To keep the problem manageable we decided to study one multi-join query. For this join query, we vary the *parallelization strategy*, the *number of processors used*, the *shape of the query tree*, and the *size of the problem*.

4.1 Test data and query

The join query studied in this performance evaluation consists of ten relations that contain equal numbers of Wisconsin tuples [BDT83]. These tuples consist of two unique integer attributes and a number of other attributes up to a total size of 208 bytes per tuple. The ten relations are joined one-by-one on their first integer attributes, and after each join they are projected to the second integer attributes and the remaining attributes of one of the operands, so that the result of each operation again is a Wisconsin relation equal in size to the operands. This test problem is similar to the problem used in [Sch90], in [ZZS93], and in [WiA93]. All possible join trees for this query have the same total execution costs. Also, the individual join operations are equal in costs and sizes of its operands. So, any differences in response time are caused by differences in the shape of the tree and the parallelization used. Therefore, such a regular tree is very suitable to study the effectiveness of the various parallelization strategies.

The test relations were generated by the PRISMA data generator. Care was taken that no correlation exists between the first and second attribute of one relations or between the unique integer attributes of different relations.

To avoid favoring one strategy, we decided to let each join query start with its ideal data fragmentation. This means that for each join query the base relations are fragmented on the join attribute of its first join over the processors that are used for this join. So, join operations that have two base-relation operands do not need redistribution of their operands prior to the join operation. The only reasonable alternative to this is starting with full fragmentation, in which all relations are fragmented over all participating processors. However, this would place SP in a special position, because that would be the only strategy to start with its ideal data fragmentation.

4.2 Experimental setup

As said before, in our experiments, we vary the *parallelization strategy*, the *number of processors used*, the *shape* of the

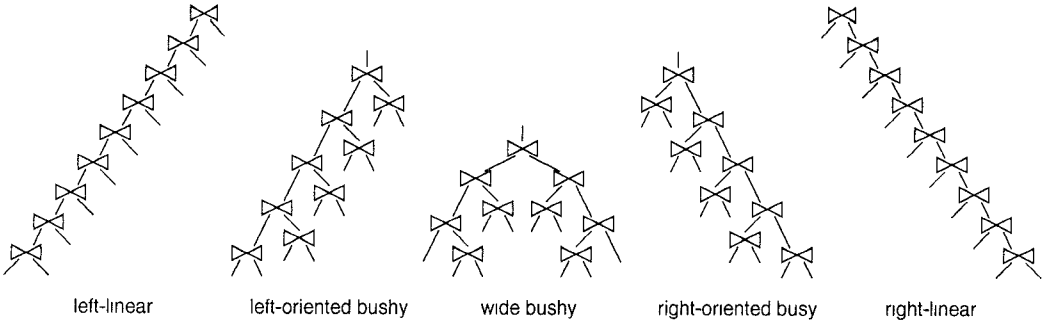

Figure 8: Query shapes used in the experiments

query tree, and the *size of the problem*. Each of three parameters is varied in the experiments. The following parameter values are chosen.

Four parallelization strategies used: SP, SE, RD, and FP. These strategies have been described above. Two problem sizes are used: the small experiment uses relations consisting of 5000 tuples each, so a total of 50000 tuples were involved in this query. The large experiment uses relations consisting of 40000 tuples each amounting to a total of 400.000 tuples in the query. These sizes will be referred to as the 5K and 40K experiments. For the 5K experiment, the number of processors used is varied from 20 to 80; for the 40K experiment we use 30 to 80 processors. The total size of the 40K query was too large to run on fewer than 30 processors. Finally, as explained in Section 3.5, we expect the strategies to perform differently for different query shapes (Figure 8). We are especially interested in the difference between (almost) linear and bushy trees, and in the difference between left and right-oriented trees. Therefore, the following 5 query shapes are used for this query: a right-linear, a right-oriented long bushy, a wide bushy, a left-oriented long bushy, and a left-linear tree.

4.3 Generation of parallel execution plans

A generator was made that can make execution plans using each of the strategies for a specific join tree. The generator takes the join tree, the cardinalities of the operand relations, the parallelization strategy, and the number of processors to be used as input, and that yields an execution plan in XRA as output.

The generator uses a cost function to calculate execution costs of the joins in the join tree. We decided to use a simple cost function for the relative costs of the individual join operations in the join tree. If $n1$ and $n2$ are the cardinalities of the join operands and r is the cardinality of the result, then the cost of a main-memory join is estimated with:

$$an1 + bn2 + cr.$$

In this formula, a and b are set to 1 if the operand is a base relation and to 2 if the the operand is an intermediate result. c is always set to 2. The rational behind this formula is as follows. The operand tuples each have to be hashed and, if the operand is an intermediate result, they have to be retrieved from the network. Result tuples have to be created and send over the network. The formula assumes that the time spent on a single action on a tuple (like hashing, retrieving from the network, sending over the network etc.) is in the same order of magnitude, which is taken as unity. Result tuples have to be created and to be sent over the network which amounts to 2 units per tuple. Operand tuples have to be hashed and to be retrieved from the network if they are from an intermediate result. This amounts to 1 or 2 units of work per operand tuple.

The cost function may seem overly simple, however, it does not seem to make sense to try and estimate the costs more precisely. As for example indicated in [SrE93], the parallelization of a query tree influences the total costs of the operations in the query tree. If a strategy allocates two operations (partially) together on one node, the transmission costs are lower than estimated. Also, parallelization may influence the need to redistribute operands between two joins. Therefore, it is in principle impossible to make a real accurate estimate of the costs of the individual join operations in the join tree. Our experiments will show, however, that the cost estimate used generates execution plans with good parallel behavior.

4.4 Results

Figures 9 through 13 show the results of the experiments for the query shapes used. In these figures, the left diagram contains the results of the 5K experiments and the right diagram contains results of the 40K experiments. Each of the figures corresponds to one query shape as indicated. In the diagrams, the response times in seconds are on the y-axis. The response times were measured as the elapsed time from the moment the scheduler starts scheduling the query until the last operation process finishes. The x-axis shows the number of processors used. Each diagram shows the results for each of the 4 strategies studied.

Left linear join tree

Figure 9 shows the results for left linear query trees. As a linear tree does not have any independent subtrees, SE allocates all available processors sequentially to each join.

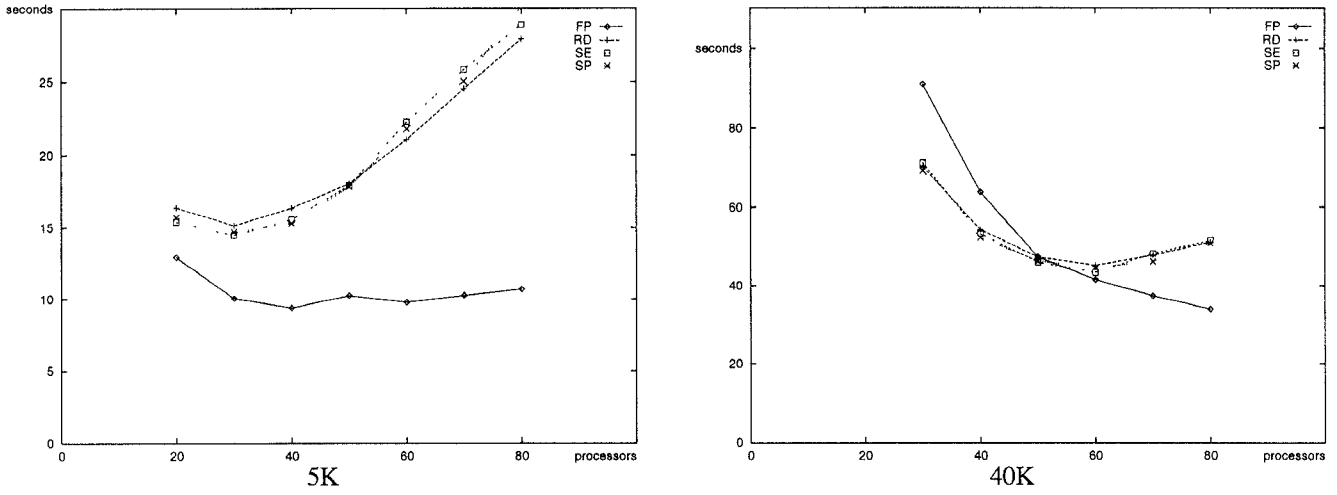

Figure 9: Left linear query tree

In this way, SE degenerates to SP for linear trees. Also, a left linear tree does not show any right-deep segments, and therefore RD allocates all available processors sequentially to each join operation. So, RD also degenerates to SP. The diagrams indeed show coinciding performance for SP, SE, and RD, both for the 5K and for the 40K experiment.

Also, it is clear that SP (and for this case also SE and RD) works reasonable for small numbers of processors, but its performance degenerates for larger numbers of processors. The 5K experiment shows this effect stronger than the 40K experiment. This performance degradation is explained by the startup costs and coordination overhead. SP needs to start one operation process for each join on each processor. So, for the 80 processor case, 800 operation processes need to be initialized. Also, the coordination overhead for redistribution of operands may be large. All intermediate results have to be refragmented. A refragmentation of n fragments into m fragments generates $n \times m$ tuple streams. So, for the 80 processor case the refragmentation of one operand generates 6400 tuples streams that have to be coordinated. The 5K experiment shows a more extensive performance degradation than the 40K experiment. This result corresponds to performance results for single operation queries (see Section 2.3.1).

FP execution of this query tree does show performance gain from parallelism. However, for the 40K experiment, its performance for a low degree of parallelism is not as good as SP. This is caused by the fact that FP suffers from the constant delay over a long linear pipeline. For a larger number of processors, FP still suffers from the delay over the pipeline, but the negative effect of startup and coordination overhead for SP is stronger. Also, for a small number of processors, FP suffers from load imbalance due to discretization errors in the distribution of processors over operations. Therefore, FP performs better for a large number of processors.

Left-oriented bushy join tree

Figure 10 shows the results for a left-oriented bushy query tree. The results for SP are similar to the results for the left linear tree. This fits with the expectation that SP is not very sensitive to the shape of the query tree. Figures 11 through 13 show similar behavior of SP for the other query shapes as well.

The results show that SE and RD work much better than for the left linear case, but not as well FP (at least not for higher numbers of processors). The shape of this query is not very suitable for either RD or SE. RD profits from independent right-deep segments, which are very short for this tree. SE profits from independent subtrees, and those are very small. As a result, there is not much room for inter-join parallelism for RD and SE. This explains why the performance of both RD and SE for this tree is in between SP and FP.

The behavior of FP is similar to its behavior for the linear tree, but a close inspection of the data shows that its performance for small numbers of processors is slightly worse than for the linear tree. This may be surprising because the pipeline for this tree is shorter than for the linear case. This result can be explained by our earlier research (see Section 2.3.3). We found that a step in a *bushy* pipeline (like the pipeline in this tree) causes a delay that is *proportional to the size of the operands.* For a low degree of parallelism, the operands of the *fragment join* are relatively large, so the delay per step in the pipeline is large. At a higher degree of parallelism the operands of the fragment join are smaller, so the delay per step in the pipeline is also smaller. This explains the relatively bad behavior for small numbers of processors and the better behavior for a larger degree of parallelism.

Wide bushy join tree

Figure 11 shows the experimental results for the wide bushy query tree. This query tree is very suitable for SE, because the tree is very wide resulting in nice independent subtrees. The results indeed show a good performance for SE. For the

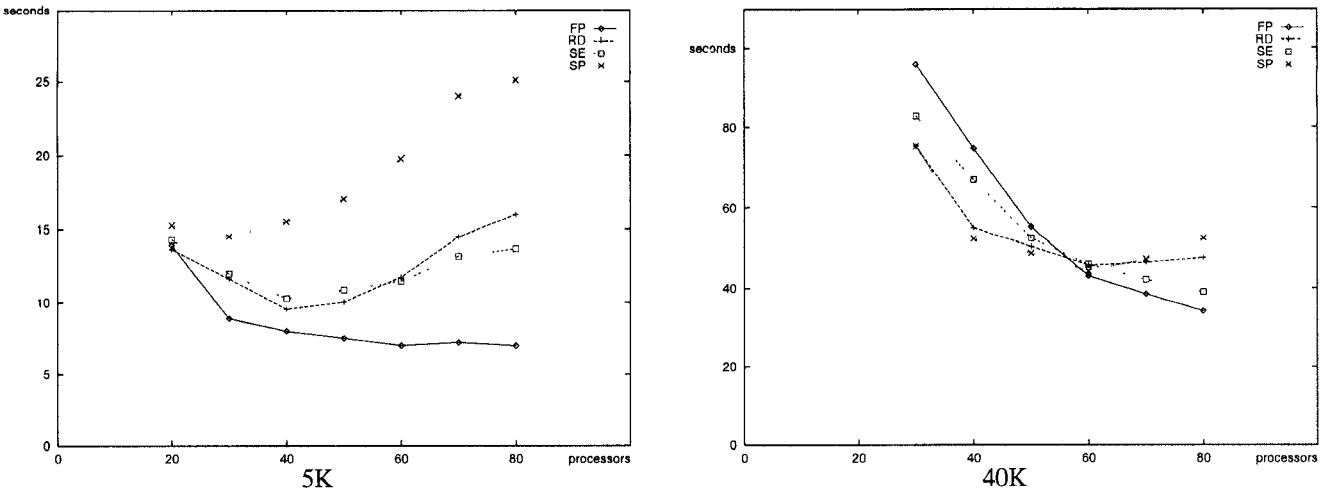

Figure 10: Left-oriented bushy query tree

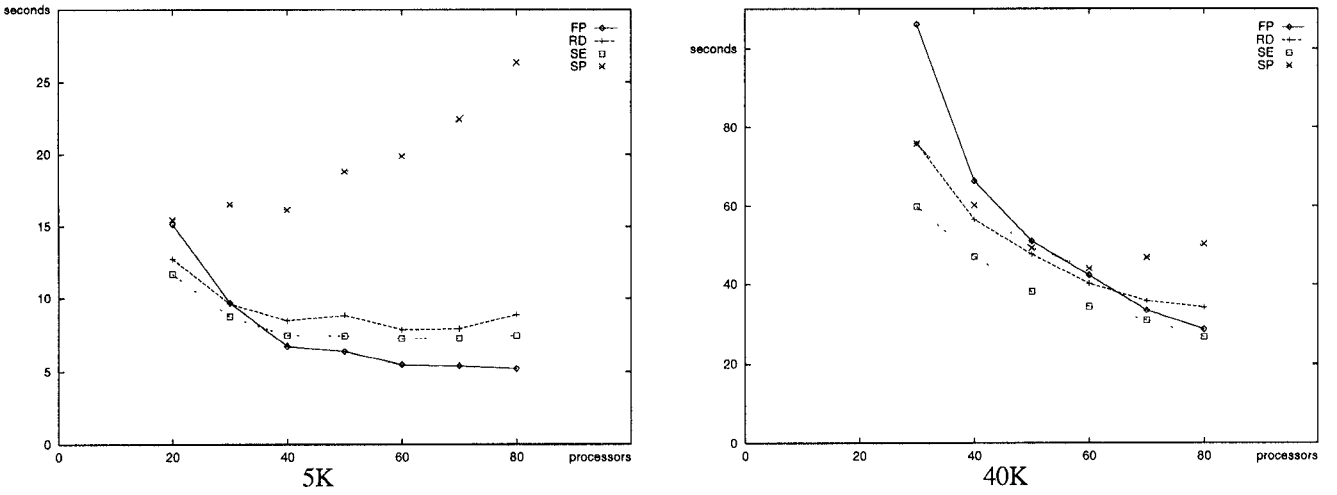

Figure 11: Wide bushy query tree

large experiment SE wins; for the small experiment SE is almost as good as FP.

FP performs well for the small experiment. This is caused by the fact that the operands are small, so FP does not suffer too much from delay over the pipeline. For a large number of operands, SE uses more operation processes than FP, so that the startup and coordination overhead dominates.

Like in the previous case FP suffers from pipeline delay for a small number of processors. This results in bad performance for a small number of processors and large operands, as explained for the previous case. Its speedup characteristics, however, outperform those of the other strategies and the performance for a large number of processors is good.

RD performs better than in the previous case, because the tree is more "right-oriented". SP performs similar to the other query shapes.

Right-oriented bushy join tree

Figure 12 shows the results for a right-oriented bushy tree. The behavior of SP is again similar. SE is not very sensitive to the orientation of the tree; its behavior resembles the behavior in Figure 10. For the same reason, FP behaves similar to the left-oriented bushy tree.

This tree is very suitable for RD. Because of the orientation to the right of this tree, a fairly long probe pipeline can be formed. The left operands for this pipeline can be processed independently in parallel on disjoint sets of processors. As a consequence, RD performs best on this tree. It should be noted however that FP performs almost as well as RD for the higher degrees of parallelism.

Right linear join tree

Finally, Figure 13 shows the results for the right linear tree. The results closely resemble th results in Figure 9, except for RD, which strategy coincides, as expected, with FP. Both

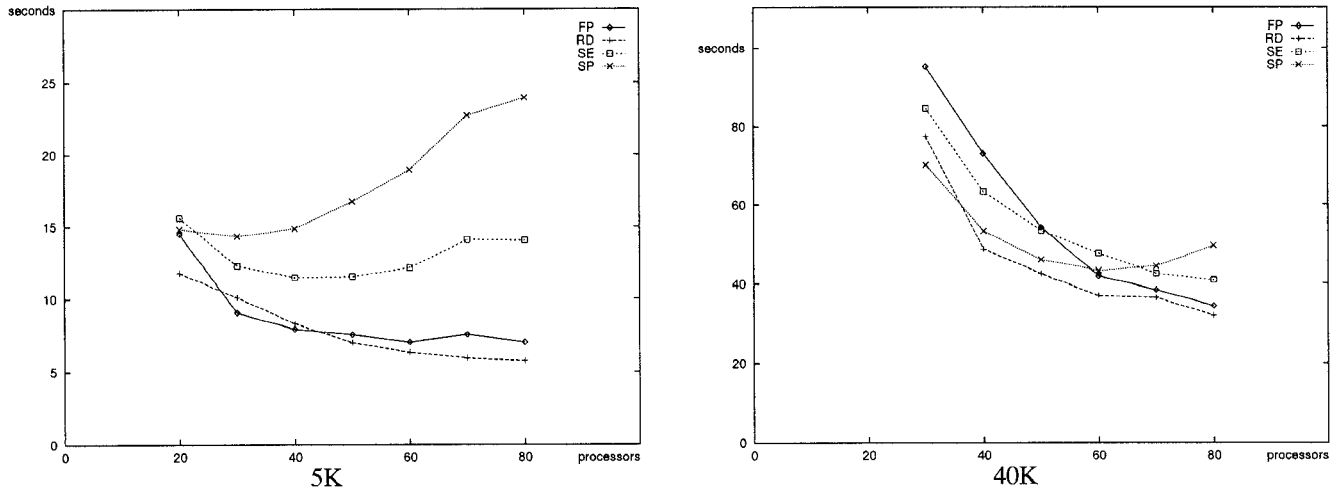

Figure 12: Right-oriented bushy query tree

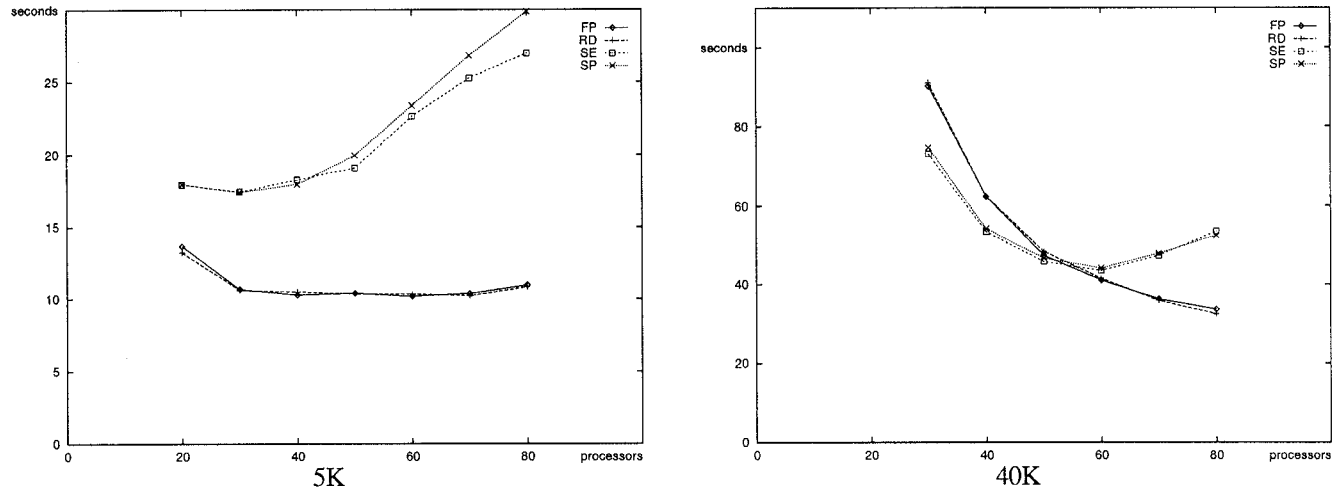

Figure 13: Right linear query tree

strategies form a linear pipeline in which all join operations process in parallel.

Because SE is not sensitive to the orientation of the tree, SE coincides with SP similar to the left linear tree.

	5K	40K
left linear	9.4 (FP40)	34 (FP80)
left bushy	7.0 (FP80)	34 (FP80)
wide bushy	5.2 (FP80)	26 (SE80)
right bushy	5.7 (RD80)	32 (RD80)
right linear	10.1 (FP60)	33 (RD80)

Figure 14: Best response times in seconds for all query trees. The strategy and the number of nodes used for the experiment that gave the results are between parentheses.

Comparison of the performance of the various query shapes

Figure 14 shows the minimal response times that were found for each query shape and for both problem sizes. This table shows that in both cases the bushy tree gives the best minimal response times. The difference is larger for the small experiment. Apparently, the delay over the pipeline of linear trees gets prohibitive. The constant delay over a linear pipeline is more important for small problems than for larger problems. The strategies that do not suffer from pipeline delay (SP and SE) use too many processors for linear trees to give good performance.

FP gives the best performance for all left-oriented trees. FP80 is the best for the 5K bushy experiment and SE80 for the 40K experiment. From Figure 11 it can be seen that FP80 gets very close to SE80 for the 40K experiment. The same remark can be made about the right-oriented trees. RD works best but FP comes very close (see Figures 12 and 13).

Disk-based systems

The experiments in this paper were using the *main-memory* DBMS PRISMA/DB. However, we feel that our results are applicable to disk-based systems as well. We do not use the PRISMA/DB assumption that the entire database should fit in the memory, we only need to be able to host the data that is related to this query. In a disk-based system with a small main-memory, which is too small to host more than a single join operation in its entirety, it will never pay off to use inter-join parallelism, because more than one join would need to share the available memory resulting in an increased disk traffic [Sch90,ZZS93]. Therefore, such systems should use SP to evaluate multi-join queries. However, in the case that the main memory of the system can host (part of) a join tree, the results presented here can be used to evaluate the fitting (sub) tree in parallel. In that case, the operands of the (sub)tree are retrieved from the disk prior to joining and after that main-memory techniques can be used to evaluate the join-tree.

5 Summary and discussion

This paper reports our experiences with the implementation of four strategies for the implementation of multi-join queries on PRISMA/DB. We have shown that PRISMA/DB provides the flexibility to implement a wide variety of parallel execution strategies for complex queries. Experiments up to 80 processors have been done. We did get performance improvement up to 80 processors. Summarizing the following conclusions may be drawn from our experiments:

- SP works fine for a small number of processors, but for a larger number of processors the startup and coordination overhead get prohibitive. The number of processors at which the overhead starts to dominate is higher for larger amount of work. For queries for which the overhead is not too large, SP is the easiest strategy, because it does not need a cost function to estimate the costs of the individual join operations. SE works well for wide bushy trees but its performance for longer trees is not very good. For linear trees, SE degenerates to SP.

- SE works very well for wide bushy trees, but its performance degenerates for more linear trees.

- RD works well for right-oriented trees. For right-linear RD degenerates to FP. For left-linear RD degenerates to SP. RD is not as sensitive to the orientation of the tree as SE is to the width of the tree. Therefore, RD works well for a wide range of query trees.

- FP gives the best overall performance over the entire range query shapes, when large numbers of processors are used. The performance for the 40K experiment on small numbers of processors is not so good, caused by discretization error and delay over the bushy pipeline (large fragment join operands).

- FP is mainly prohibited by pipeline delay. For bushy trees this overhead decreases with an increasing number of processors. SP, and to a lesser extent RD and SE, are prohibited by startup and coordination overhead, which increases with an increasing number of processors. Therefore, FP is expected to eventually yield the best performance on bushy trees if more processors are added.

- Bushy trees give better performance results than linear trees.

From the results clear guidelines for the parallel implementation of multi-join queries can be formulated. For a *small number of processors*, Sequential Parallel execution (SP) is the easiest and best way to evaluate a multi-join query in parallel. For *larger numbers of processors*, Full Parallel execution (FP) performs quite well. SE and RD perform well for differently sized problems, but only on suitable query shapes. FP, SE, and RD need a cost function to estimate the costs of the constituent binary joins in the tree. If there it is possible to choose between a linear and a *bushy* tree with (almost) equal processing costs, the bushy one should be chosen, because bushy trees allow more effective parallelization.

RD does not work too well for trees that contain left-deep segments. However, it is possible without cost penalty to mirror (parts of) a query to make it more right-oriented, so that in practice RD is expected to work quite well. It should be noted that RD uses less memory than FP because only one hash-table needs to be built.

FP gives the best performance results for small queries. This means that it allows a large degree of parallelism on a relatively small queries. This is caused by the fact that the overhead for this strategy is relatively small and also the main overhead decreases with an increasing number of processors. From this observation, we expect FP to do the best job in scaling up to even larger numbers of processors than used in this paper.

The experiments reported in this paper are done using a regular query on a synthetic database. It would be quite interesting to use the strategies presented here for real-life applications.

References

[Ame91] P. America, ed., *Proc of the PRISMA workshop on parallel database systems*, Springer-Verlag, New York–Heidelberg–Berlin, 1991.

[ABF92] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten & A. N. Wilschut, "PRISMA/DB: A parallel, main-memory, relational DBMS," *IEEE Transactions on Knowledge and Data Engineering* 4 (December 1992), 541–554.

[ApW94] P. M. G. Apers & A. N. Wilschut, "Understanding large scale parallelism for data management ," in *Keynote at the 3rd PDIS Conf, Austin, Texas, USA, Septemb er 1994.*

[BDT83] D. Bitton, D. J. DeWitt & C. Turbyfill, "Benchmarking database systems - A systematic approach," in *Proc 9th VLDB Conf, Florence, Italy, October 31–November 2, 1983.*

[BAC90] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith & P. Valduriez, "Prototyping Bubba, A highly parallel database system," *IEEE Transactions on Knowledge and Data Engineering* 2 (1990), 4–24.

[BrG89] K. Bratbergsengen & T. Gjelsvik, "The development of the CROSS8 and HC16-186 (Database) computers.," in *Proc of 6th IWDM workshop, Deauville, France, June 1989*, 359 –372.

[CaK92] F. Carino & P. Kostamaa, "Exegesis of DBC/1012 and P-90 industrial supercomputer database machines," in *PARLE, Paris, France, June 1992*, 877–892.

[CLY92] M. S. Chen, M. L. Lo, P. S. Yu & H. C. Young, "Using segmented right-deep trees for the execution of pipelined hash-joins.," in *Proc 18th VLDB Conf, Vancouver, Canada, August 23–27, 1992*, 15–26.

[CYW92] M. S. Chen, P. S. Yu & K. L. Wu, "Scheduling and processor allocation for parallel execution of multi-join queries.," in *Proc 8th Data Engineering Conf, Tempe, Arizona, USA, February 3–7, 1992*, 58–67.

[DGS90] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao & R. Rasmussen, "The GAMMA database machine project," *IEEE Transactions on Knowledge and Data Engineering* 2 (March 1990), 44–62.

[DeG92] D. J. DeWitt & J. Gray, "Parallel database systems: The future of high-performance database systems," *Communications of the ACM* 35 (June 1992), 85–99.

[Gre92] P. W. P. J. Grefen, *Integrity control in parallel database systems*, PhD-Thesis, University of Twente, 1992.

[GWF91] P. W. P. J. Grefen, A. N. Wilschut & J. Flokstra, "PRISMA/DB1 User Manual," Memorandum INF91-06, Universiteit Twente, Enschede, The Netherlands, 1991.

[HoS91] W. Hong & M. Stonebraker, "Optimization of parallel query execution plans in XPRS," in *Proc 1st PDIS Conf, Miami Beach, Florida, USA, December 1991.*

[HWF93] M. A. W. Houtsma, A. N. Wilschut & J. Flokstra, "Implementation and performance evaluation of a parallel transitive closure algorithm on PRISMA/DB," in *Proc 19th VLDB Conf, Dublin, Ireland, August 24 - 27, 1993*, 206–217.

[HCY94] H. I. Hsaio, M. S. Chen & P. S. Yu, "On parallel execution of multiple pipelined hash-joins," in *Proc of ACM-SIGMOD, Minneapolis, MN, May 24–27, 1994*, 185–196.

[KBZ86] R. Krishnamurty, H. Boral & C. Zaniolo, "Optimization of nonrecursive queries," in *Proc 12th VLDB Conf, Kyoto, Japan, August 25–28, 1986*, 128–137.

[LVZ93] R. S. G. Lanzelotte, P. Valduriez & M. Zait, "On the effectiveness of optimization search strategies for parallel execution strategies," in *Proc 19th VLDB Conf, Dublin, Ireland, August 24 - 27, 1993*, 493–504.

[LST91] H. Lu, M. C. Shan & K. L. Tan, "Optimization of multi-way join queries for parallel execution," in *Proc 17th VLDB Conf, Barcelona, Spain, September 3–6, 1991*, 549–560.

[Sch90] D. Schneider, "Complex query processing in multiprocessor database machines, PhD thesis," Computer Sciences Technical Report 965, Universiteit of Wisconsin, Madison, USA, September 1990.

[ScD89] D. Schneider & D. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," in *Proc of ACM-SIGMOD, Portland, OR, May 31–June 2, 1989*, 110 – 121.

[ScD90] D. A. Schneider & D. J. DeWitt, "Tradeoffs in processing complex join queries via hashing in multiprocessor database machines," in *Proc 16th VLDB Conf, Brisbane, Australia, August 13–16, 1990*, 469–480.

[SAC79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie & T. G. Price, "Access path selection in a relational database management system," in *Proc of the ACM-SIGMOD 1979, Boston, USA*, 82 –93.

[SHV92] M. Spiliopoulou, M. Hatzopoulos & C. Vassilakis, "Using parallelism and pipeline for the optimization of join queries," in *PARLE, Paris, France, June 1992*, 279–294.

[SrE93] J. Srivastava & G. Elsesser, "Optimizing multi-join queries in parallel relational databases.," in *Proc 2nd PDIS Conf, San Diego, California, USA, January 1993*, 84–92.

[SKP88] M. Stonebraker, R. Katz, D. Patterson & J. Ousterhout, "The design of XPRS," in *Proc 14th VLDB Conf, Los Angeles, CA, August 29–September 1, 1988*, 318 – 330.

[SwG88] A. Swami & A. Gupta, "Optimization of large join queries.," in *Proc of ACM-SIGMOD, Chicago, IL, June 1–3, 1988*, 8 –17.

[Wil93] A. N. Wilschut, *Parallel query execution in a main-memory database system.*, PhD-Thesis, University of Twente, 1993.

[WiA91] A. N. Wilschut & P. M. G. Apers, "Dataflow query execution in a parallel, main-memory environment," in *Proc 1st PDIS Conf, Miami Beach, Florida, USA, December 1991*, 68–77.

[WiA93] A. N. Wilschut & P. M. G. Apers, "Dataflow query execution in a parallel, main-memory environment," *Journal of Distributed and Parallel Databases.* 1 (1993), 103–128.

[WiA90] A. N. Wilschut & P. M. G. Apers, "Pipelining in query execution," in *Proc of the International Conference on Databases, Parallel Architectures and their Applications, Miami, USA, March 1990.*

[WAF91] A. N. Wilschut, P. M. G. Apers & J. Flokstra, "Parallel query execution in PRISMA/DB," in *Proc of the PRISMA Workshop on Parallel Database Systems, Noordwijk, The Netherlands, September 1990*, P. America,ed., Springer-Verlag, New York–Heidelberg–Berlin, 1991.

[WFA92] A. N. Wilschut, J. Flokstra & P. M. G. Apers, "Parallelism in a main-memory system: The performance of PRISMA/DB.," in *Proc 18th VLDB Conf, Vancouver, Canada, August 23 - 27, 1992.*

[WiG93] A. N. Wilschut & S. A. van Gils, "A model for pipelined query execution," in *Proc of the MASCOTS93 symposium, January 1993, San Diego, California.*,

[ZZS93] M. Ziane, M. Zait & P. borla-Salamet, "Parallel query processing in DBS3," in *Proc 2nd PDIS Conf, San Diego, California, USA, January 1993*, 93–102.