Distributed and Parallel Databases 1 (1993), 103–128

Dataflow Query Execution in a Parallel Main-Memory Environment

ANNITA N. WILSCHUT ANNITA@CS.UTWENTE.NL
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

PETER M.G. APERS APERS@CS.UTWENTE.NL
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract. In this paper, the performance and characteristics of the execution of various join-trees on a parallel DBMS are studied. The results of this study are a step into the direction of the design of a query optimization strategy that is fit for parallel execution of complex queries.

Among others, synchronization issues are identified to limit the performance gain from parallelism. A new hash-join algorithm is introduced that has fewer synchronization constraints than the known hash-join algorithms. Also, the behavior of individual join operations in a join-tree is studied in a simulation experiment. The results show that the introduced Pipelining hash-join algorithm yields a better performance for multi-join queries. The format of the optimal join-tree appears to depend on the size of the operands of the join: A multi-join between small operands performs best with a bushy schedule; larger operands are better off with a linear schedule. The results from the simulation study are confirmed with an analytic model for dataflow query execution.

Keywords: parallel query processing, multi-join queries, simulation, analytical modeling

1. Introduction

During the last years much attention has been paid to the development of parallel DBMSs. Using special purpose hardware has shown not to be successful; instead, a parallel DBMS running on general purpose shared-nothing hardware appears to be the right choice [8]. Also, various query processing strategies have been implemented: dataflow query processing appears to be superior to control-flow scheduling of queries [9, 19]. Therefore, this paper studies query processing in a general purpose, shared-nothing, dataflow architecture.

Teradata [20], GAMMA [9], Bubba [5], HC16-186 [6], and PRISMA [1] are examples of parallel DBMSs that actually were implemented. Each of these systems exploits some sort of parallelism to speed up query execution. Within a query, interoperator and intraoperator parallelism can be discriminated [17, 23, 26]. Orthogonal to this distinction, pipelining can be contrasted to (pure) horizontal parallelism. This last type is called parallelism here, like in many other papers. *Intraoperator* parallelism is the primary source of parallelism in the projects mentioned above. This type of parallelism is well understood now, and using it, efficient execution strategies can be found for simple queries. The Wisconsin benchmark [3], which consists of such simple queries on large volumes of data is used to describe the performance of a system [6, 9].

A dataflow architecture, however, offers the possibility to also exploit *interoperator* parallelism and pipelining by allocating different relational operations to different (sets of) processors. The potential of using different types of parallelism for one query, turns query optimization into a difficult problem, that cannot be solved using conventional query-optimization techniques, due to the large number of execution plans that is possible for one query. So far, little research has been done in this research area although it is identified to be important for the further development of parallel DBMSs [5, 8]. The query optimizers for most parallel DBMSs are based on the theory developed in [18], however, this theory is not particularly fit for parallel dataflow query processing. For example, only *linear* query trees are considered, although this class of trees does not necessarily include the optimal one for a parallel environment.

In a first attempt to understand the effect of various query tree formats, [17] studies the behavior of right-deep and left-deep linear query trees for multi-join queries. It is concluded in that paper, that right-deep scheduling has performance advantages in the context of GAMMA. In [10] it is shown how arbitrarily shaped query trees can be parallelized using the "exchange" operator, which splits a (part of) a query tree into a number of subtrees that can be executed in parallel. Although that paper makes clear that certain query trees can be parallelized, it does not solve the problem of which (type of) query tree performs best.

In this paper, we study the execution of multi-operation queries. The ultimate goal of this study is the design of a query optimizer for a parallel DBMS. As we chose to study the execution of large complex queries, this query optimizer should aim at reducing the response time of these complex queries, rather than optimizing the throughput for some workload. Relational multi-join queries are used as an example, because the join is an important, and expensive relational operation. An outline of the path, we want to follow for this research is as follows: Query optimization comes down to selecting an execution strategy with low costs [14]. Because searching the entire space of possible strategies is not feasible, most query optimizers are heuristic [4, 12]. Heuristics are based on *insight* in the essentials of query execution. So, to design a heuristic query optimizer for a parallel DBMS, it is essential to *understand* the behavior of execution strategies for a query on a parallel DBMS. Modeling is a way to gain insight in the essentials of parallel query execution. Two approaches to modeling were used in our study: simulation and analytical modeling. Both approaches allow studying the response time of different execution strategies, and the utilization of participating processors. The resulting knowledge, should eventually lead to formulation of query optimization heuristics. It should be emphasized that we do not aim at a detailed quantitative model for a parallel DBMS, but rather at a simple, understandable framework that, by its nature, yields *insight* in the modeled phenomenon. However, neither intuition, nor the sort of model that is presented here can *validate* the heuristics that they yield, and so, these heuristics have to be validated against a real dataflow DBMS, or a detailed simulation if the former is not available. The methodology described in this paragraph, is similar to the methodology that is common in science: Scientists try to understand natural phenomena by modeling them. Subsequently, hypotheses are formulated, that are based on the model, and these hypotheses are validated against reality by experimentation.

This paper describes a first step on the path outlined above. First, the results of a simulation study are described. This work resulted in the proposal of a new join-algorithm that is fit for dataflow query execution. Also, the execution characteristics of multi-join queries were studied. An attempt to fully understand the results of this simulation study led to the development of an analytical model. The first results of this analytical model confirm the results of our simulation study. In the near future, we want to follow the path by extending the analytical model. That step should lead to the formulation of query optimization heuristics.

The research reported in this paper is carried out in the context of PRISMA/DB [1, 2, 13, 26]. PRISMA/DB is a parallel, main-memory, relational DBMS that runs on a 100-node shared-nothing architecture. The implementation of PRISMA/DB was finished in 1991, and [25] evaluates its performance. The fact that PRISMA is a *main-memory* system plays an important role in our research. The price of primary memory has fallen sharply during the last years. As this trend is expected to continue, an interesting question arises: How can huge amounts of memory be used? In this study this question is specialized into: Can a very large primary memory yield performance gain in a DBMS? Therefore, we are willing to accept using large amounts of memory, if performance gain is expected in return.

The remainder of this paper is organized as follows: The next section describes dataflow query execution in a main-memory environment. Section 3 presents simulation results. Section 4 introduces the analytical model and its elaboration for join operations and join trees. The last section summarizes and concludes the paper.

2. Dataflow query execution

A main-memory parallel DBMS running on shared-nothing hardware has the following features: The hardware consists of a number of processors that can communicate via a message-passing network. Each processor hosts part of the base-data. A processor can access its part of the base-data directly. If a processor wants to access the data stored on another processor, the processor storing the data has to send the data to the processor that needs it via the network.

A query on a relational database can be represented as a dataflow graph. The nodes of such a graph represent eXtended Relational Algebra operations [11]. Leaf nodes have base relations as operand, intermediate nodes work on intermediate results. Each processor can run one or more operations processes. In this paper, we want to study interoperator parallelism, and therefore each operation process is assumed to have a private processor. This assumption implies that intermediate results have to be transported via the network to another processor. Operation processes evaluate XRA-operations on their local data, or on tuple streams that are sent to them via the message-passing network. The result of the evaluation of an XRA-operation consists of a (multi)set of tuples. Such a result can either be stored locally, in which case it can be accessed by the local professor later on; or it can be sent to one or more other operation processes. In the last case, the sending and the receiving operation processes can run concurrently, forming a pipeline.

Network transport of tuples is modeled as follows: To transport a tuple from a process to another, remote process, first, it has to be "wrapped" and put on the network hardware by the sending operating system, then, it is sent over the network, and finally, it has to be retrieved from the network and "unwrapped" by the receiving operating system. So, sending a tuple over the network implies CPU costs on the sending and receiving processor, and actual transmission, which implies a delay. In general, the CPU costs involved, appear to be the limiting factor, and, therefore, the *rate* at which tuples are transported over the network is determined by the capacity of the CPUs that send and receive the tuples and not by the capacity of the network hardware. So, tuple transport is modeled in terms of CPU costs on two processors [8] and a constant transmission delay.

3. Simulation of dataflow query execution

This section describes the results of a simulation study. First, the simulation program used is described, then we study join algorithms, and finally we describe a simulation of multi-join queries.

3.1. The simulator

To study the execution characteristics of a query, a simulator for parallel query execution was developed. This was done, because at the time at which this research was started, PRISMA/DB was not ready yet. Also, the simulator is a flexible tool to study parallelism. The input to the simulator is a schedule for a query. In such a schedule, the size, fragmentation and allocation of the operands and the intermediate results can be specified. The output consists of a diagram for each operation process that was used in the schedule. These diagrams plot the processor utilization (on behalf of that operation process) against the time. The (horizontal) time-axis can be scaled. Figure 1 shows an example of such a diagram.

The simulator models local processing and network transport of tuples. The local processing model uses simple cost formulas for relational operations. Network transport of tuples is modeled as CPU-costs on the sending and receiving

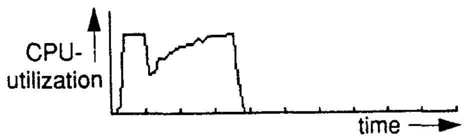


Figure 1. Sample output of the simulator.

processor according to the description above. The simulator is parameterized with the costs of simple operations on tuples. Most important, the ratio between the costs of local processing, and the CPU-costs related to network transport of tuples is set by the parameters. The parameter values that were used in this paper are measured from PRISMA/DB.

3.2. Join algorithms

The choice of a join algorithm influences the execution characteristics of a multi-join query in different ways.

Firstly, the processing, I/O, and communication costs are influenced. Schneider and DeWitt [16] give an overview of well-known join algorithms and evaluate their performance for simple join-queries by experimentation. Hash-join algorithms are shown to be the most efficient ones for equi-joins. Therefore, in our paper, only hash-join algorithms are considered.

Secondly, the synchronization between the joins that participate in a more complex join query, is determined by the join algorithm used. In this section, the synchronization requirements of a well-known hash-join algorithm are studied. Because those requirements are too tight to allow considerable performance gain from pipelining, a new main-memory hash-join algorithm is proposed that has fewer synchronization requirements [22].

The known hash-join algorithms, grace hash-join, simple hash-join, and hybrid hash-join, are disk-based, and they only differ in the way disks are used. Therefore, only one main-memory version of these algorithms is dealt with in this paper. This algorithm is called simple hash-join here.

3.2.1. ***Simple hash-join.*** The simple hash-join algorithm consists of two phases (see Figure 2). In the first phase, one entire operand is read into an in-memory hash-table. In the second phase, the tuples of the other operand are read one by one, each tuple is hashed and compared to the tuples in the corresponding bucket in the hash-table of the first operand. If a match is found, an output tuple is produced. This algorithm is asymmetric in its operands, although the join-operation is conceptually symmetric. The result is only formed during the second phase of the algorithm.

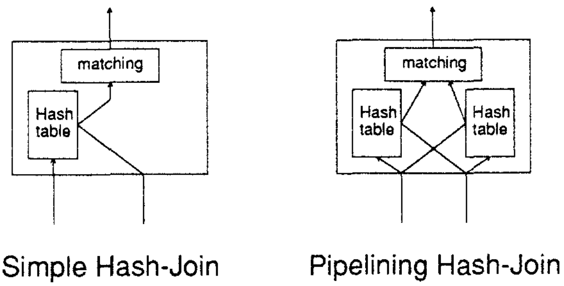


Figure 2. Simple hash-join and pipelining hash-join algorithm.

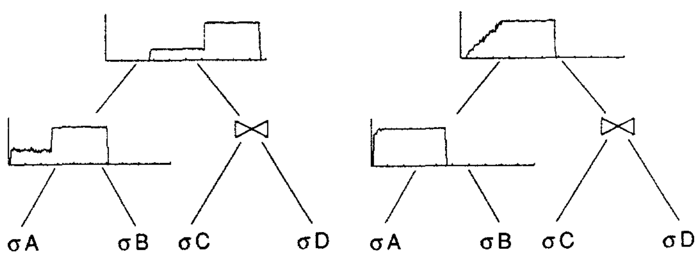


Figure 3. Simple hash-join and pipelining hash-join.

3.2.2. ***Pipelining hash-join.*** The pipelining hash-join algorithm (see Figure 2), aims at producing its output tuples as early as possible in the process of calculating the join, without decreasing the performance of the join operation itself. During the join process a hash-table for both operands is built. 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. Finally, the tuple is inserted in the hash-table of its own operand. When the last tuple of one of the operands is processed, the join process can stop building a hash-table for the other operand, because this hash-table will not be used any more. Keeping this last feature in mind, it is easy to see that the pipelining hash-join degenerates to a simple hash-join when one operand is available to the join process entirely, before the first tuple of the other operand arrives. The pipelining hash-join algorithm is symmetric in its operands.

3.2.3. Evaluation of the simple hash-join and the pipelining hash-join.

Figure 3 shows the execution characteristics of the simple hash-join processes, and of the pipelining hash-join processes in a four-way multi-join

$$(\sigma A \bowtie \sigma B) \bowtie (\sigma C \bowtie \sigma D)$$

as visualized by the simulator. The figure shows the join tree for this query; two join symbols in each tree are replaced by the diagrams showing the execution characteristics of the corresponding join processes. As explained in Section 3.1, these diagrams plot the processor utilization of the processor executing the join against the time. Because the characteristics of $\sigma A \bowtie \sigma B$ and $\sigma C \bowtie \sigma D$ are identical, only one of these join symbols is replaced by a diagram. The time axis in all diagrams is scaled to the response time of the query using the simple hash-join algorithm.

The join processes read their *input* from selection processes, that produce output at a limiting rate. This was done to make the distinction between the two phases of the simple hash-join visible. The pipelining hash-join makes a faster start than the simple hash-join, because tuples belonging to both operands can be processed right from the beginning. Also, the pipelining hash-join starts producing *output* earlier than the simple hash-join. So, the consumer of the result of the pipelining hash-join can start earlier than the consumer of simple hash-join. As a result, the response time of the evaluation with the pipelining hash-join is better.

The CPU-utilization of the pipelining hash-join is increasing in time. This is caused by the increasing probability to find matching tuples as the hash-tables are filled. For join-operations that are higher up in a join tree this effect is enlarged by the fact that the operand tuples arrive at the join process with increasing rate.

The difference in synchronization requirements described above shows that the pipelining hash-join allows more interoperator pipelining than the simple hash-join, and so it fits more naturally in a dataflow execution model.

It is the asymmetry in the simple hash-join algorithm that explains the difference between left-deep, and right-deep scheduling reported in [17]. Using a symmetric algorithm, like the pipelining hash-join yields the same performance for any linear join-tree. In the next section, the behavior of linear join-trees, and other join-tree formats is studied.

3.3. *Multi-join queries*

In this section, the trade-offs of using differently structured query trees for the execution of multi-join queries are discussed. Also, the performance of the simple, and the pipelining hash-join in multi-join queries are compared. Figure 4 shows a linear and a bushy join tree for an eight-way multi-join query.

3.3.1. Trade-offs in join-tree formats. First, some terminology is introduced. The term *hop* is used for the transmission from one join operation to its parent (the operation consuming its output) or its operand. The *termination delay* over one or more hops is the difference in termination time of the adjacent join operations. The term *delay* is used as a shorthand for termination delay.

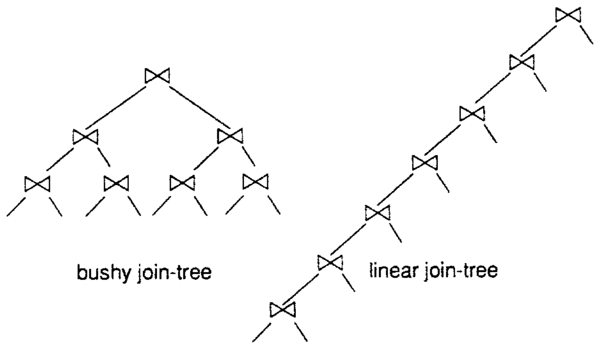


Figure 4. A bushy and a linear join tree.

Various types of nodes in a join tree can be identified.

- The leaf-nodes in a tree have two base relations as operands. These base-relations are available to the join process immediately.
- The intermediate nodes in a linear tree and some intermediate nodes in bushy trees have one base relation and one intermediate result as operands. The base relation is available to the join process immediately, but the join process has to wait for the other operand to become available from the previous join-operation.
- Bushy join trees contain join-processes that have two intermediate results as operands. Such a join process has to wait for both operands, and, therefore, the join process does not start immediately.

Having to choose a join tree for a join query, we are faced with the following trade-off: The join processes in a linear join tree can all start immediately hashing their base-relation operand; in this way, they fill the time waiting for the other operand. On the other hand, a linear join tree contains the longest possible pipeline, causing a larger number of delays on top of each other. The pipelines in a bushy join tree are shorter than the ones in a linear one, but some intermediate join processes have to wait for both their operands, what may lead to large delays. In the next section, an experiment is described that shows that the optimal format of the join tree for a multi-join depends on the size of the operands.

3.3.2. An experiment. To study the execution characteristics of various join trees, a join between 16 relations that have equal numbers of equally sized tuples, matching one tuple in each operand to exactly one tuple in another operand, is studied. The tuples that result from a join operation are projected to the size of the tuples in the operands. As the size of the tuples is equal throughout the query, the size of the operands is determined by the number of tuples in them. All join operations have a private processor. All possible join trees for this query yield the same amount of joining and data communication costs. Also, the individual joins in the query are equal in costs, and sizes of their operands. So, any differences in response time are caused by differences in the synchronization of the join tree that is used only.

In four subexperiments, the 16-way join described above is evaluated for operands with resp 1000, 5000, 10,000, and 50,000 tuples. The response times of those queries are measured with the simulator for a linear, and a symmetric completely bushy tree. The response times are shown in Table 1.

Table 1. Response times in seconds.

		1000	5000	10000	50000
pipelining hash-join	bushy	3.9	16.2	31.1	149.0
	linear	6.8	13.9	23.0	95.4
simple hash-join	bushy	5.5	22.5	43.8	214.0
	linear	7.0	15.4	25.8	109.5

First, we want to remark, that the pipelining algorithm outperforms the simple hash-join in all cases. The difference in performance is larger for bushy scheduling than for linear scheduling. This is caused by the fact that the pipelining hash-join degenerates to the simple hash-join in join operations that are relatively close to the root of a linear tree. In those join operations, the entire base-relation operand is processed before the tuples of the other operand are available. In the remainder of this section, only the schedules using the pipelining hash-join are considered.

Apparently, the bushy scheduling performs better for small operands and linear scheduling is better for large operands. Figures 5 and 6 show the execution characteristics of linear and bushy query trees for small and large operands. These figures show the join trees that were used (see Figure 4). Similar to Figure 3, some join symbols are replaced by the simulator diagrams of the corresponding join-processes. In each join tree, the time-axis of the diagrams is scaled to the response time of the corresponding query.

From the diagrams in Figure 5, we see that in a *linear* tree, there is a *constant* termination delay over the pipeline between two adjacent joins. This delay does not depend on the number of tuples in the operands. Some diagrams show two distinct phases in the processing of the join. The first phase is the construction of a hash-table for the base-relation operand. The other phase is joining the other

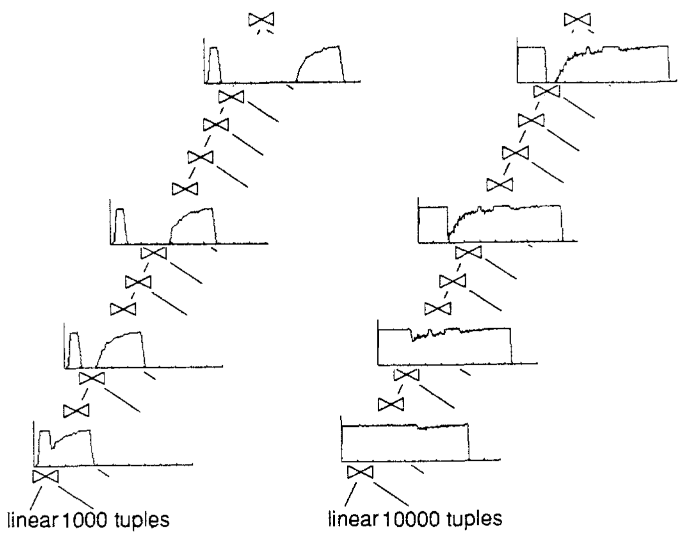
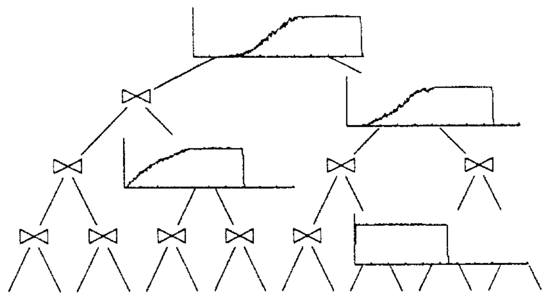


Figure 5. Execution characteristics of linear join trees.

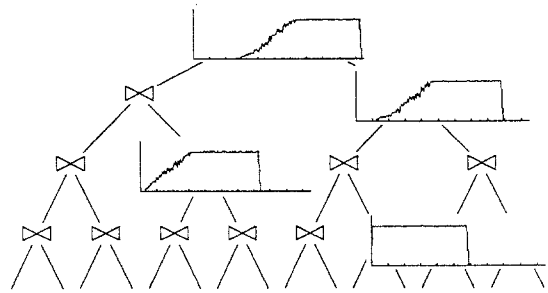
operand to this hash-table. In the lowermost join operation these two phases are mixed, as described by the pipelining hash-join algorithm. The further up in the join tree, however, the longer the join-operation has to wait for its second operand. At a certain point, the complete hash-table for the base-relation is read before the first tuple of the second operand arrives. From this point on, the pipelining hash-join behaves similar to the simple hash-join. The point at which the two operands of the join are processed completely separately is reached earlier for small operands than for larger ones.

The diagrams for the bushy tree lead to the following observations. The leaf-nodes show the same characteristics as the leaf-node of the linear tree. The delay over one hop is larger than in case of a linear tree, because neither operand is directly available. Within one query each hop yields approximately the same delay. Moreover, a closer look at the characteristics shows that the scaled diagrams are similar for the large and the small query. This means that the entire experiment scales with the number of tuples in the operand, and therefore, the delay over one hop is *proportional* to the size of the operands. This proportionality is a surprising result that cannot be accounted for intuitively.

The difference in termination delay between linear and bushy trees can explain the fact that bushy trees work better for small operands, and linear trees for larger operands. In both cases the response time of the entire query is equal to the sum of the execution time of a leaf-node join operation, and the accumulated termination delay in the query. For both the linear and bushy tree, the execution time of a leaf-node join operation is proportional to the number of tuples in one



bushy 10000 tuples



bushy 1000 tuples

Figure 6. Execution characteristics of bushy join trees.

operand. This part of the response time is equal for a linear and a bushy tree with equal operands. The accumulated termination delay, however, is the same for all linear trees, and proportional to the number of tuples in the operands for bushy trees. So, the accumulated termination delay for bushy trees grows linearly with the number of tuples in the operands. The constant accumulated termination delay for a linear tree is larger than the (small) delay of the bushy tree for small operands, but smaller than the (large) delay the bushy tree with large operands. So, at a certain operand size, the linear trees outperform the bushy ones. With the parameter setting that was used in our simulation experiment, the break-even point lies at about 2000 tuples.

The next section introduces a mathematical model for dataflows query section that can explain the simulation results for bushy join-trees.

4. Analytical modeling of dataflow query execution

In this section, an analytical model for dataflow query execution is developed. The key idea behind this model is the fact that the *rates* at which tuples are transported, and processed in a dataflow system are modeled. As such, the model views a query in execution as an "assembly line," in which the tuples are the items to be transported along the operations processes which serve as workers. The operation processes map the rate at which tuples are available to them onto the rate at which they produce output tuples. This mapping depends on the type of operations process, and on the resource (CPU capacity) that is available.

First, a general model shows how an abstract relational operation maps its input stream onto an output stream. The general model can be specialized to model specific relational operations. In this paper, we describe the model for the pipelining hash-join. Other relational operation, however can fairly easily be modeled also [21]. Linking the models for individual join operations together yields a model for a join-tree. That model can explain the surprising proportional termination delay in bushy query trees, that was found in the previous section.

4.1. Some preliminaries

4.1.1. Resources in the model. The model describes the *rates* at which tuples are transported and processed in a dataflow system. Also, the utilization of the processors participating in the dataflow system is modeled. Because, as described above, the bandwidth of the message-passing network is assumed to exceed the requirements of the application, the utilization of this hardware is not modeled. This paper only deals with retrieval, and in a main-memory context, retrieval does not need any disk-accesses. So, there is no need to model secondary storage either. The only resource that has to be taken into account now is the CPU. The resulting model is simple and consequently powerful: a complete analysis is possible for some classes of queries.

4.1.2. Modeling discrete phenomena. Tuples are discrete entities. Our model, however, is continuous. A continuous model for a discrete phenomenon is possible if large numbers of events are described [15, 24]. The transition from a discrete to a continuous model eliminates the need to use probability theory; if, in a discrete model, there is a probability 0.5 that a tuple is generated, the continuous model will generate half a tuple. This way of modeling has generally been accepted in physics and biology, and can be used here without problems.

4.1.3. Entities and dimensions. The rate at which tuples are transported and processed, and the utilization of processors are modeled. To do so, the costs of certain operations are expressed. Tuple transport is expressed in number of tuples per timeunit. The processor load is dimensionless, and has a maximum of 1. The costs of operations are expressed in time units per tuple. Consider as an example, an operation that processes tuples at rate x tuples/timeunit. The processor spends A timeunits for the processing of one tuple. The resulting processor load is Ax (dimensionless).

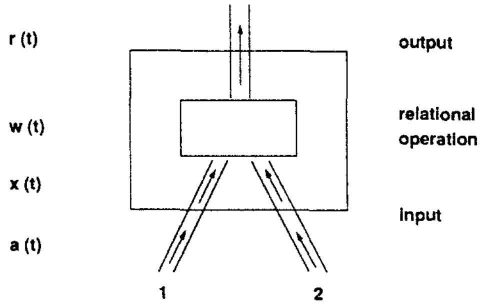


Figure 7. Formalism used to develop an analytical model for one data flow operation.

4.2. Definition of a dataflow model

Figure 7 summarizes the essentials of a dataflow operation. The large box in this figure represents the processor; the small box represents the operation process. Data is sent to the operation process at the bottom of the box; the result is sent away at the top of the box.

4.2.1. Terminology. Each operation process has one or two operands. In this paper, only the join, which is a binary operation is considered, so there are always two operands. In Figure 7, each input stream contains two arrows: the first arrow indicates the rate at which tuples are *available* to the operations process, the second one represents the rate at which tuples are *processed* by the operation process. The arrow in the output stream indicates the rate at which result tuples are produced. The left column in Figure 7 shows the formalism used:

$a(t)$ is the rate at which tuples of a particular operand are available to an operation process at time t.
$x(t)$ is the rate at which tuples of a particular operand are processed by an operation process at time t.
$w(t)$ is the processor utilization at time t.
$r(t)$ is the rate at which tuples are produced at time t.

The functions $a(t)$ and $x(t)$ can be labeled with a subscript to indicate which operand is meant.

A query in execution, consists of a number of communicating operation processes. Time $t = 0$ is used to indicate the starting point of the *entire* query. Some operation processes in the query may be idle at time $t = 0$. T is used for the time at which an operation process is ready.

4.3. Some relationships

From the description of dataflow query execution, the following relationships can be deduced:

An operation process cannot process tuples in an operand before they have arrived at its input stream. This can be modeled as

$$\forall t \ in \ [0, T] \ : \int_0^t a(\tau)\ d\tau \geq \int_0^t x(\tau)\ d\tau$$

The processor utilization is a function of the rate at which the operand tuples are processed:

$$w(t) = \mathcal{W}(x_1(t), x_2(t)) \tag{1}$$

The rate at which tuples are produced is also a function of the rate at which the operand tuples are processed:

$$r(t) = \mathcal{R}(x_1(t), x_2(t))$$

If a CPU works at full capacity, its utilization is 1. Therefore, $w(t)$ can never be larger than 1.

Our model discriminates between the rate at which operand tuples are available, and the rate at which they are processed. This is done, because these two may differ, if the operation process cannot keep up with the rate at which tuples are sent to it. This observation leads to the definition of two different modes in which an operation process can work.

input-limited mode Tuples are sent to the operation process at such low rate, that the operation process can keep up with this rate. Now, $w(t) \leq 1$ and $x_j(t) = a_j(t)$ for both operands.

CPU-limited mode Tuples are sent to the operation process at such high rate, that the receiving processor cannot keep up with this rate, so $w(t) = 1$ and there is an operand for which $x_j(t) < a_j(t)$.

This discrimination leads to the central equation in this paper:

$$\begin{array}{ll} x_j(t) = a_j(t) & \text{if process input–limited} \\ x_j(t) \text{ meets } w(t) = 1 & \text{if process CPU–limited} \end{array} \tag{2}$$

This equation is used to evaluate the behavior of an operation process. An outline of such an evaluation is as follows: Equation (1) expresses the CPU-utilization as a function of the rates at which operand streams are processed. In the input-limited mode of an operation process $x_j(t) = a_j(t)$, and $\mathcal{W}((a_1(t), a_2(t)) < 1$. In CPU-limited mode, the join operation cannot keep up with rate at which tuples arrive so $\mathcal{W}((a_1(t), a_2(t)) > 1$. Therefore, evaluation of $\mathcal{W}((a_1(t), a_2(t))$, and comparing the result to 1 (the maximal CPU-utilization) can reveal whether an operation process is input-limited, or CPU-limited at time t. If an operation process is input-limited, the rate at which operand streams are processed is clear ($a(t)$). If, on the other hand, the process appears to be CPU-limited, then solving equation $\mathcal{W}((x_1(t), x_2(t)) = 1$, for $x_j(t)$ shows at what rate each operand tuple-stream is processed.

Knowing the functions $x_j(t)$ and mapping $\mathcal{R}$, the rate at which tuples are produced by an operation process, $r(t)$, can be calculated. The result of an operation process can be sent as input to another operation process. Those tuples are assumed to arrive at the receiving operation process with some delay[1] at the rate at which they are produced by the producing operation process. So, then function $a(t)$ for a consumer is known, and we are in the position to evaluate the behavior of this consumer process.

Summarizing, the model maps the rate at which operand tuples are available, to the rate at which result tuples are produced. To describe a query tree, the result of the evaluation of one operation can be used as input to a next one.

In the remainder of this section, the model developed above is specialized to describe the pipelining hash-join. After that, the results of this evaluation are used to study bushy join-trees. In all cases, the goal is full characterization of the participating operations in terms of $x(t)$, $w(t)$, $r(t)$, and T, given the rate at which operand tuples are available ($a(t)$).

4.4. Pipelining hash-join

In this section, it is assumed that the operand tuples from both operands are available at a nonlimiting rate, so, the operation process only processes in its CPU-limited mode. The more general case, in which the join-process works both input-limited, and CPU-limited, is dealt with afterwards. Both operands are equal in size: they contain n tuples. The selectivity of the join operation is assumed to be ϱ: the result contains ϱn^2 tuples.

From the description of the pipelining join algorithm, it is clear that the join algorithm processes tuples from both operands at the same rate ($x(t)$). The goal of this section is finding $x(t)$, and deriving T, and $r(t)$ from $x(t)$.

During the join operation, each operand tuple has to be made available to the join-process, its hash-value has to be calculated, it has to be inserted in a hash-table and it has to be compared to the tuples in the corresponding bucket of the other operands hash-table. These costs are assumed to be constant during the join-process (A).[2] If a match is found, a result tuple is generated. The costs associated with producing one tuple (concatenation, projection, storage or network transport) are assumed to be constant too (S). The distinction between work dedicated to processing operand tuples and to generating result tuples is essential. Through this distinction, insight is gained in how an operation process maps its input streams onto an output stream.

4.4.1. The model. The development of the model for a join operation derives an equation for the load of the processors as a function of the rate at which the operand tuples are processed. Solution of this (integral) equation yields the maximum rate at which a join process can process its input.

The tuples in both operands are processed at the same rate ($x(t)$). Note, that this rate is time dependent. The model will reveal that this rate decreases with the time. So, the amount of work the processor spends on processing input tuples is equal to

$$2Ax(t)$$

There is a factor 2 in this expression, because tuples from both operands are processed.

The amount of work spent on generating the result is calculated as follows: The number of tuples in the entire first operand that join with one tuple in the second operand is equal to

$$\varrho n$$

Therefore, the number of tuples that a tuple, arriving at time t, matches with is proportional to the number of tuples that have already arrived in the other operand. The number of tuples that have arrived in an operand at time t is equal to

$$\int_0^t x(\tau)\ d\tau$$

So, the number of result-tuples that is formed upon the arrival of one tuple at time t is equal to

$$\varrho \int_0^t x(\tau)\ d\tau$$

Using this expression, the amount of work spent on generating the result can be formulated as

$$2\varrho S x(t) \int_0^t x(\tau)\ d\tau$$

Again, the factor 2 is caused by the fact that tuples from both operands are processed.

The CPU-utilization is equal to the sum of the amount of work spent on processing the input, and the amount of work spent on generating the output.

$$w(t) = 2Ax(t) + 2\varrho S x(t) \int_0^t x(\tau)\ d\tau \tag{3}$$

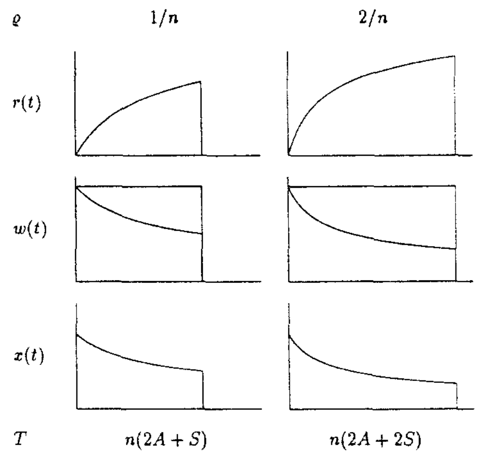


Figure 8. Execution characteristics of two join operations as produced by the analytical model.

The join-operation process in its CPU-limited mode, so equation (2) can be specialized to:

$$x(t) \quad \text{meets} \quad 2Ax(t) + 2\varrho Sx(t) \int_0^t x(\tau)\ d\tau = 1 \tag{4}$$

4.4.2. Finding the rate at which operand tuples are processed. We are now ready to find $x(t)$ from equation (4). This integral equation can be solved using elementary calculus [21]:

$$x(t) = \frac{1}{2\sqrt{\varrho St + A^2}} \tag{5}$$

The bottom row of Figure 8 shows two diagrams that plot $x(t)$ against t. As expected, $x(t)$ decreases in the time, because, as the join processes proceeds, more effort has to be spent on generating result tuples, because the hash-tables are filled.

4.4.3. Termination of the join-process. The join operation is ready at time T. At this time, n tuples per operand have been processed:

$$\int_0^t x(\tau)\ d\tau = n$$

Substitution of equation (5) and then solving this equation for T yields:

$$T = 2An + \varrho Sn^2 \tag{6}$$

This result is reasonable: $2n$ operand tuples and ϱn^2 result tuples have to be processed. This processing costs $2An + \varrho Sn^2$ units of time. The CPU-utilization is equal to 1 during the entire join-process, so, the join process will end at $2An + \varrho Sn^2$.

4.4.4. The output stream. The rate at which result tuples are produced can be derived from equation (4):

$$r(t) = 2\varrho x(t) \int_0^t x(\tau)\, d\tau = \frac{1 - 2Ax(t)}{S}$$

Substitution of (5) yields

$$r(t) = \frac{1}{S} \left\{ 1 - \frac{A}{\sqrt{\varrho St + A^2}} \right\} \qquad (7)$$

The top row of Figure 8 shows diagrams that plot $r(t)$ against the time. $r(t)$ is increasing in time, because the probability of finding a match in the join-process increases with the time.

4.4.5. Two join-processes with different selectivity. Figure 8 shows some diagrams of the characterization of two different join-processes.[3] Two join-operations are illustrated: one in which the selectivity of the join was chosen to be $1/n$ so that the result contains n tuples, and another in which the selectivity of the join is $2/n$, so that the result contains $2n$ tuples. The first join-operation is in the first column of diagram in the figure, and the second join-operation is in the second.

The topmost diagrams show the rate at which result tuples are generated as a function of the time. This rate in increasing in the time due to increasing probability of finding a match. At the very beginning of the join-operation $r(t)$ is equal to zero, due to the fact that no tuples to form a match with have arrived yet. As expected, the second join-operation produces tuples at a higher rate than the first one.

The middle diagrams show the processor-utilization as a function of the time. As the join-operations are CPU-limited during the entire operation, the processor utilization is equal to 1. In these diagrams, an additional curve shows what portion of the CPU-effort is spent on processing input tuples (area below the curve), and what proportion is spent on generating output (other area). We see that the less selective join-operation spends a larger portion of its effort on generating output, and that in both cases the amount of work related to generating output increases, at the expense of processing input.

The bottom diagrams show the rate at which input tuples are processed. As expected this rate is decreasing, and this effect is stronger for the second join-operation.

The model developed in this section, yields an analytical expression for the rate in which output tuples are produced (equation (7)). In the next section the output stream of a single join-operation is used as input to a next join-operation: in this way join trees are studied.

4.5. Symmetric bushy join trees

The behavior of symmetric bushy join trees is analyzed. Figure 4*a* shows a symmetric tree for an eight-way join. It is assumed that the operands are equal in size (each operand contains n tuples) and that the join operations match one tuple in their left operand to exactly one tuple in their right operand, so the selectivity (ϱ) of the join-operations is equal to $1/n$. Each join-operation has a private processor. The bushy join tree in the simulation experiment that was described in Section 3 corresponds to the sort of join trees that are modeled here. It is clear that the join-operations that have two base relations as operands, all have the same execution characteristics. These join operations are called level0 joins. The join operations that join the results of level0 joins again have the same characteristics. They are called level1 joins. In the same way, level2, level3 and even higher levels can be defined.

4.5.1. The model. Here, the models for individual join-operations, like the one developed in the previous section are linked together to describe a bushy join tree.

It is assumed, that the base operands are available to the level0 joins at nonlimiting rates. Therefore, the characteristics of these joins are as described in a previous section. To describe the other levels, some notation conventions are needed. In this section, subscripts are used to indicate the level of the join-operation. Due to the symmetry of the problem, we do not need to discriminate between the two input streams of one join-operation. So, $x_1(t)$ denotes the rate at which tuples of either level1 join-operand are processed.

The level0 joins operate in their CPU-limited mode from the beginning on. Higher level joins, however, are expected to show an increase in their CPU-utilization: they start input-limited, and after some time they switch to CPU-limited. The following symbols are used to describe this:

- θ_i The time at which a CPU executing a level$i$ join is saturated. So, at this time, the join process switches from its input-limited to its CPU-limited mode. θ_0 is equal to 0.
- $\mathcal{H}_i$ The number of tuples that have been collected in the hash-table of an operand of a level$i$ join at time θ_i. $\mathcal{H}_0$ is equal to 0 and $\mathcal{H}_i = \int_0^{\theta_i} x(\tau)\, d\tau$.

Result tuples from one level are sent as input to a join-operation in the next level. So,

$$a_{i+1}(t) = r_i(t) \tag{8}$$

Now, we can derive the model for a join-operation at level$i$ of a bushy join tree. Equation (8) is used to characterize the input rate, and the central equation of this paper (2) is used to model the operation process. Furthermore, an expression for CPU-utilization (3) for the pipelining join is derived in the previous section. The combination of these three equations, and the definition of θ_i yields the model for level i of a bushy join tree:

$$\begin{array}{lcl} x_i(t) & = & r_{i-1}(t) \text{ if } 0 \le t < \theta_i \\ x_i(t) & \text{meets} & 2Ax_i(t) + 2\frac{S}{n}x_i(t)\int_0^t x_i(\tau)\,d\tau = 1 \quad \text{if } t \ge \theta_i \end{array}$$

In these equations $r_i(t)$ is defined by

$$r_i(t) = \frac{2}{n}x_i(t)\int_0^t x_i(\tau)\,d\tau$$

Finding the rate at which operand tuples are processed. These equations can be solved explicitly for $x_i(t)$ [21]. Level0 was solved in the previous section:

$$x_0(t) = \frac{1}{2\sqrt{\varrho S_j t + A_j^2}} \tag{9}$$

For $i > 0$, $x_i(t)$ can expressed recursively as

$$x_i(t) = \begin{cases} r_{i-1}(t) & \text{if } 0 \le t \le \theta_i \\ x_{i-1}(t-\delta) & \text{if } t > \theta_i \end{cases} \tag{10}$$

where

$$r_i(t) = \frac{2}{n}x_i(t)\int_0^t x_i(\tau)\,d\tau \tag{11}$$

Also it can be derived that

$$\theta_i = \sigma + i\delta \tag{12}$$

where σ and δ are constants that are *proportional* to n. Finally it can be concluded that

$$\mathcal{H}_i = \mathcal{H}_j, \quad i, j > 0 \tag{13}$$

Note that, although this solution is formulated recursively, it is well-defined, because it is initialized with x_0. This solution looks rather complex, but it can be interpreted in the following way.

- Two phases can be discriminated in a join-process: The *startup* phase, in which the join-process does not saturate its processor, and the *main* phase, in which the join-process is CPU-limited. In a level0 join the startup phase has length 0 (so, there is no startup phase).
- The main phases of subsequent join levels are similar, apart from a translation δ in time. This implies that the main phase of each subsequent level starts and ends δ time units after its predecessor.
- The startup phase of a join-operation takes longer for higher levels, but the number of tuples that are processed ($\mathcal{H}$), and consequently the amount of work that is done during startup is equal for each level.

Termination of the processes. In the previous section the termination time for level0 joins was derived:

$$T_0 = 2An + \varrho Sn^2 \tag{14}$$

Because each subsequent level does the same amount of work during its startup phase, and its main phase is translated δ with respect to its predecessor, it is easy to see that

$$T_i = T_0 + i\delta \tag{15}$$

From this result, it can be concluded that the termination delay of subsequent join levels is proportional to n. Also the response time of the entire query is proportional to n. These statements confirm the surprising results from our simulation study (see Section 1).

*4.5.2. **Examples.*** Figure 9 show diagrams that plot $w(t)$ against the time for level0 through level3. Similar to the diagrams for $w(t)$ in Figure 8, an additional plot in the diagrams shows that portion of the CPU-time is spent on processing the input. Figure 9 has two columns of diagrams: the first column is on a 16-way join between operands of 1000 tuples; the second one shows the same join with operands of 1500 tuples. The time axis of all diagrams is scaled in the same way. Comparison of the diagrams in one column shows that the termination delay between subsequent levels is constant. Comparison of all diagrams shows that the termination delay is proportional to the number of tuples in one operand.

*4.5.3. **Consequences for optimization and scheduling of multi-join queries.*** Although the model for join trees has to be extended to cover general join trees, it can now already be indicated how the sort of results that the model yields (with the results

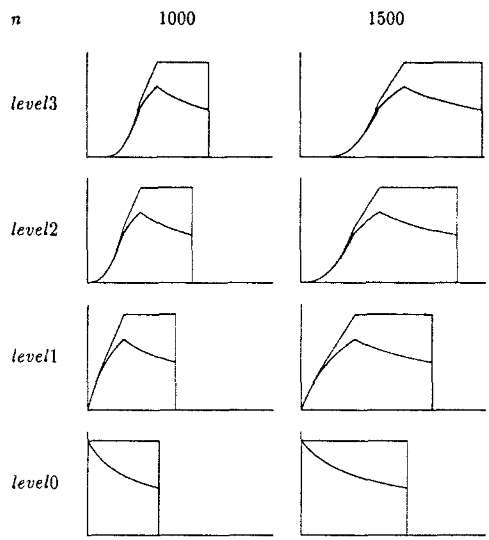


Figure 9. Execution characteristics of join-operations in a bushy schedule.

for symmetric bushy trees as an example), can be used for query optimization, and query scheduling. Apparently, the results not only indicate the *amount* of work that has to be done on behalf of one relational operation; they also give an indication of *when* this work has to be done and how busy a processor will be. The following examples illustrate how this scheduling information can be used.

- A query optimizer, having to select an execution strategy for a query, can use both the amount of work that has to be done and the timing information. It is possible that a more expensive schedule (in terms of total processing costs) has very good timing characteristics, so that its response time is very good. If response time is the important figure in the system, such a schedule should be selected.
- The knowledge about when processors are busy can be used by a scheduler: A processor that is assigned to a relational operation can be used for other purposes (possibly another relational operation) during the time that it is idle.
- The model for bushy query trees shows that in the execution of a join, a startup, and a main phase can be discriminated. If the scheduling of a join at level$i$ is postponed until time $i\delta$, the main phase of the join is left the same, and the startup phase uses the processor at full capacity during a shorter period of time. Therefore, postponing the scheduling of a level$i$ join until time $i\delta$ does not affect the response time of the entire query. In Figure 10, the characteristics of a level2 join are shown. The left diagram shows the characteristics of a join that is scheduled immediately after query startup, the

Figure 10. Scheduling of join-operations in a bushy tree.

right one shows the characteristics when the join is scheduled at time 2δ.

Although a limited type of join trees was modeled in this section, we feel that the model increased our insight in the working of the pipelining join algorithm, and its cooperation with its producers and consumers. Specifically, the fact that the model can predict when the higher level join-operation needs CPU capacity is encouraging. Also, the fact that postponing the scheduling of certain join-operations to some extent does not influence the response time to the entire query can be used easily in a scheduler: two different relation operations can be scheduled subsequently to one processor.

5. Conclusions and future work

The work reported in this paper is a part of our research on query optimization strategies for a parallel dataflow DBMS. In the introduction to this paper, a methodology to do this research was outlined. It was illustrated that gaining understanding of parallel dataflow query processing is an essential step. This understanding should used to design a heuristic query optimizer for a parallel dataflow DBMS.

The work reported in this paper was introduced as a step into the direction of understanding parallel dataflow query execution. Looking back, we are faced with some questions: "What insight was gained from our study?", "How can this knowledge be used?", and "How can this knowledge be validated?". These questions are now answered in turn.

What did we learn from our study?

- The simulation study showed that different aspects of the algorithms which are used for relational operations in a query tree are important. Apart from, of course, the CPU-costs of an algorithm, also its synchronization with the processes that produce and consume its input and output is important to yield a good performance. It has shown that the well-known simple hash-join algorithm has synchronization requirements that are too tight to allow performance gain from pipelining. A new hash-join algorithm, the pipelining hash-join, was proposed that is expected give good performance in a dataflow system. Algorithms for other relational operations can be studied in the same way. Pipelining algorithms are possible for many relational operations.

- The simulation study of various join-tree formats gave insight in the behavior of individual join-processes in relationship to their position in a join tree. It was shown that the time at which a join-operation can start processing depends on the position of the join-operation in the join tree, and on the sizes of the operands. Regular linear, and bushy trees were studied extensively. Other join-tree formats need additional study.
- The mathematical model which was developed in this paper confirms some of the simulation results. Also, the model can predict the effect of changes in the scheduling of join-processes, as indicated in the previous section.

How can this knowledge be used? The gained insight can be used in several ways. Firstly, pipelining algorithms should be used in dataflow systems. Secondly, a query optimization design, based on the ideas developed in this paper becomes feasible. Knowing the CPU-costs, and the delays that are incurred in a join tree, the response time to a join-query can be calculated. This cost evaluation can be used in combination with known query optimization techniques, that search (part of) the space of possible execution strategies to find the cheapest one. Finally, the insight in the timing requirements can be used in a scheduler, as illustrated in Figure 10.

How do we validate this knowledge. Currently, we are planning experiments on the latest version of PRISMA/DB.

Following the path of our research requires the study of more general join trees. Some preliminary work in this direction has been done and the results are encouraging. We plan to incorporate other relational operations in our model, and we want to study the effect of distributing individual relational operations. Although the model was only evaluated for a limited class of queries, we can now already make statements about the scheduling of operations in query trees. It should be emphasized that, as explained in the introduction, these statements have to be validated.

This paper is about parallel query execution. The concrete results of our study are worthwhile, and they probably eventually will be used to design new query execution strategies. We feel, however, that apart from the concrete results, the approach to obtaining them also is a contribution to the database research. The experimental approach which is adopted from science, combined with mathematical modeling of the observed phenomena, is, to our opinion, a viable methodology to tackle certain problems in computer science.

Notes

1. The transmission delays can easily be handled in our model, and their influence on the results is simple. The formalism is complicated by using them however,

and also, the transmission delay is assumed to be small compared to the time needed to evaluate a relational operation. Therefore, we choose not to incorporate them in the model in this paper.

2. Actually these costs are increasing slightly during the join process, due to the fact that hash-buckets are filled. Using a good hash-table though, minimizes this increase.
3. Part of the symbolic manipulation and the generation of plots of the results of this manipulation was carried out using the symbolic manipulator Maple [7].

References

1. P. America (ed.), *Proc. PRISMA Workshop Parallel Database Systems*, Springer-Verlag: New York, 1991.
2. P.M.G. Apers, C.A. van den Berg, J. Flokstra, P.W.P.J. Grefen, M.L. Kersten, and A.N. Wilschut, "PRISMA/DB: A parallel main-memory relational DBMS." To appear in IEEE transactions on Knowledge and Data Engineering.
3. D. Bitton, D.J. DeWitt and C. Turbyfill,"Benchmarking database systems—A systematic approach," in M. Schkolnick and C. Thanos (eds.), *Proc. 9th Int. Conf. Very Large Data Bases, Florence, Italy*, VLDB Endowment: Saratoga, CA, 1983.
4. P. Bodorik and J.S. Riordon,"Heuristic algorithms for distributed query processing," in S. Jajodia, W. Kim and A. Silberschatz (eds.), *Proc. Int. Symposium on Databases Parallel Distributed Systems, Austin, Texas*, IEEE Press: Montvale, NJ, pp. 107–117, 1988.
5. H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez,"Prototyping Bubba, a highly parallel database system, *IEEE Trans Knowledge Data Eng.*, Vol. 2, no. 2, pp. 4–24, 1990.
6. K. Bratbergsengen and T. Gjelsvik,"The development of the CROSS8 and HC16-186 (Database) computers," in H. Boral and P. Faudemay (eds.), *Proc. 6th Int. Workshop Database Machines*, Deauville, France, June 1989, Springer-Verlag: New York, pp. 359–372, 1989.
7. B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monager, and S.M. Watt, *Maple Reference Manual*, WATCOM: Waterloo, Canada, 1988.
8. D.J. DeWitt and J. Gray,"Parallel database systems: The future of database processing or a passing fad?," *ACM SIGMOD Record*, vol. 19, no. 4, pp. 104–112, 1990.
9. D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen,"The GAMMA database machine project," *IEEE Trans. Knowledge Data Eng.*, vol. 2, no. 1, pp. 44–62, 1990.
10. G. Graefe,"Encapsulation of parallelism in the volcano query processing system," in H. Garcia-Molina, H.V. Jagadish (eds.), *Proc. ACM-SIGMOD 1990 Int. Conf. Management Data, Atlantic City, NJ*, ACM Press: New York, pp. 102–111.
11. P.W.P.J. Grefen, A.N. Wilschut, and J. Flokstra,"PRISMA/DB1 User Manual," Universiteit Twente, Enschede, The Netherlands, Memorandum INF91-06, 1991.
12. M. Jarke and J. Koch,"Query optimization in database systems," *Comput. Surv.*, vol. 16, no. 2, pp. 111–152, 1984.
13. M.L. Kersten, P.M.G. Apers, M.A.W. Houtsma, H.J.A. van Kuijk, and R.L.W. vande Weg,"PRISMA : A Distributed main memory database machine," in *Proc. 5th Inter. Workshop Database Machines, Karuizawa, Japan,* 1987.
14. E. van Kuijk,"Semantic query optimization in distributed database systems," Ph.D. thesis, University of Twente, 1991.
15. A. Okubo, *Diffusion and Ecological Problems: Mathematical Models*, Springer-Verlag: New York, 1980.

16. D.A. Schneider and D.J. DeWitt,"A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," in J. Clifford, B. Lindsay and D. Maier (eds.), *Proc. ACM-SIGMOD 1989 Inter. Conf. Management Data, Portland, OR,* ACM Press: New York, 1989 (Also appeared as ACM SIGMOD Record, vol. 18, no. 2, 1989.)
17. D.A. Schneider and D.J. Dewitt,"Tradeoffs in processing complex join queries via hashing in multiprocessor database machines," in D. McLeod, R. Sacks-Davis and H. Schek (eds.), *Proc. 16th Int. Conf. Very Large Data Bases, Brisbane, Australia*, Morgan Kaufmann: Palo Alto, CA, pp. 469–480, 1990.
18. P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie and T.G. Price,"Access path selection in a Relational Database Management System," in *Proc. ACM-SIGMOD 1979 Int. Conf. Management Data*, Boston, MA, pp. 82–93, 1979.
19. W.B. Teeuw and H.M. Blanken,"Control versus data flow in distributed database machines," Universiteit Twente, Enschede, The Netherlands, Memorandum INF91-02, 1991.
20. Teradata Corporation,"Teradata,"DBC/1012 Database Computer Concepts and Facilities," C02-0001-00, 1983.
21. A.N. Wilschut,"A model for dataflow query execution in a parallel main-memory environment," Universiteit Twente, Enschede, The Netherlands, Memorandum INF91-34, 1991.
22. A.N. Wilschut and P.M.G. Apers, "Pipelining in query execution," in N. Rishe, S. Navathe, and D. Tal (eds.), *Proc. Int. Conf. Databases, Parallel Architectures and their applications, Miami,* IEEE Press: Montvale, NJ, 1990.
23. A.N. Wilschut, P.M.G. Apers, and J. Flokstra,"Parallel query execution in PRISMA/DB," in P. America (ed.), *Proc. PRISMA Workshop Parallel Database Systems, Noordwijk, The Netherlands,* Springer-Verlag: New York, 1991.
24. A.N. Wilschut and P.G. Doucet,"Theoretical studies on animal orientation: A model for kinesis," *Theoret. Biol.* vol. 127, pp. 111–125, 1987.
25. A.N. Wilschut, J. Flokstra, and P.M.G. Apers,"Parallelism in a main-memory system: The performance of PRISMA/DB.," in *Proc. 18th Int. Conf. Very Large Data Bases, Vancouver, Canada,* 1992.
26. A.N. Wilschut, P.W.P.J. Grefen, P.M.G. Apers, and M.L. Kersten, "Implementing PRISMA/DB in an OOPL.," in H. Boral and P. Faudemay (eds.), *Proc. 6th Int. Workshop Database Machines, Deauville, France,* Springer-Verlag: New York, pp. 359–372, 1989.