Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

On Optimal Pipeline Processing in Parallel Query Execution

S. Manegold, F. Waas, M.L. Kersten

Information Systems (INS)

INS-R9805 February 28, 1998

Report INS-R9805
ISSN 1386-3681

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

On Optimal Pipeline Processing in Parallel Query Execution

Stefan Manegold Florian Waas Martin L. Kersten

CWI
P.O.Box 94079, 1090 GB Amsterdam, The Netherlands
⟨*firstname.lastname*⟩*@cwi.nl*

ABSTRACT

A key assumption underlying query optimization schemes for parallel processing is that their cost models can deal with the multitude of effects encountered during the execution phase. Unfortunately, this is rarely the case and the optimal processing is only achieved in a few situations.

In this paper we address the problem to achieve optimal processing under a pipelined execution strategy. The approach taken is based on a novel analytical framework—which establishes a formal treatment of both dataflow and processing environment—to validate execution strategies. The framework is based on the notion of π-optimality which reflects an execution strategy's ability of ad-hoc resource utilization. π-optimal strategies are insensitive to skew and provide a transparent interface to parallelism as they ensure a provable near-optimal exploitation of the processing environment.

Finally, we discuss several strategies and present a π-optimal execution strategy. Experiments carried out on an SMP verify our considerations: The new algorithm outperforms conventional pipelining execution substantially and is resistant against various kinds of skew.

1991 Computing Reviews Classification System: [H.2.4] Parallel Database Systems, Query Processing
Keywords and Phrases: parallel and distributed databases, parallel query processing, dynamic load balancing, efficient resource utilization
Note: Funded by the HPCN/IMPACT project.

1. Introduction

Query optimization in parallel database systems is, following a common approach, split into two phases [11, 7]: sequential optimization and parallelization. The former involves query rewrites and join ordering to arrive at an optimal sequential *query evaluation plan (QEP)*. The latter deals with mapping a sequential QEP to a parallel execution environment. The final result is a parallel query execution plan (cf. Fig. 1).

Much research has been devoted to achieve the best possible parallelization of a given sequential plan [8, 9, 13, 3, 10]. A common approach is to incorporate many features of the target architecture in the cost model, e.g. communication costs or hardware description. Based on this information a static parallel schedule is derived [8, 4]. However, from a validation point of view increasing the number features considered during optimization is dangerous. The prime reason is that small errors in the estimates propagate through a QEP. Such estimation errors turn out to be exponential [12] and lead to suboptimal parallel schedules. This situation aggravates when the underlying data is skewed.

To sum up, the execution of an optimized parallel plan is very likely to diverge form the envisioned optimum. The straight forward solution to enrich the optimizer with better statistics is generally not feasible as their maintenance costs are too high.

Figure 1: Two-phase optimization approach

To reduce the complexity of the optimizer and to achieve stable results, a transparent way of using a parallel execution environment is sought—i.e. once the optimizer determined the degree of parallelism for a certain QEP, the execution strategy has to ensure maximal resource utilization.

In this paper we develop a means to transparently exploit parallelism. We give a formal model based on two components: dataflow and processing model. The dataflow model handles the relational algebraic operators and their data dependencies in a uniform way. The processing model abstracts a parallel environment as queues and processing units. Guided by this model we discuss the principles to find an optimal solution and introduce the notion of π-optimal as a measure of ad-hoc exploitation of pipelining parallelism. We call a processing strategy π-optimal if a processing unit is only idle when no unit of work can be assigned to them at this very moment. We prove that the execution time of a π-optimal strategy is at most twice the theoretical optimum. Therefore, π-optimal strategies provide the transparent interface to the optimizer.

We use our model to analyze conventional pipelining execution strategies, as given in [8] and DTE, a data parallel algorithm [15, 14]. DTE outperforms PE in most cases significantly. However, it is not yet π-optimal: in typical skew situations, some processors may become overloaded while others remain idle. We present DTE/R an improved variant of the previous algorithm which overcomes this problem by redistribution of the intermediate processing results. In contrast to the other algorithms mentioned, DTE/R is π-optimal. The redistribution adds only little overhead, but pays back significantly in almost every case of skew.

To verify our approach we present a comprehensive quantitative assessment of a prototypical implementation. Our results confirm the effectiveness of π-optimality. It shows that our analytical worst-case bound is rather pessimistic compared to the average performance observed.

Related Work. The problem of scheduling queries on parallel environments has attracted a lot of attention. Hasan and Motwani point out the importance of pipelining parallelism and develop near optimal scheduling heuristics with respect to minimize communication overhead [8]. Their techniques apply to a restricted class of query plans, such as star queries and paths of pipelined operators. The heuristics proposed ignore skew handling as well as intra-operator parallelism. Chekuri et al. develop a more general treatment and allow for arbitrary query plans using the same cost model [2]. Again, skew is not considered. Garofalakis and Ioannidis discuss a richer cost model and focus on shared-nothing architectures [4, 5]. Their scheduling heuristics are also based on the assumptions that no skew affects the execution. Lo et al. study constraint processor allocation for pipelined hash joins [13, 3] and extend this approach in [10] to scheduling of separate pipelining segments. For their simulation model, they assume uniformly distributed attributes and exclude skew situations. In addition to our effort, Bouganim et al. have independently obtained a partly π-optimal algorithm, similar to DTE [1].

Road-Map. In Section 2, we develop a dataflow model that reflects the relational algebraic structure of the query to be processed. Section 3 describes the processing model to capture the

parallel processing environment. The basic principles of the algorithms and an analysis, based on the two models, is given in Section 4. The results of our experiments are presented and discussed in Section 5. Section 6 concludes the paper.

2. Dataflow Model

In this Section, we develop an abstract *dataflow model* to reflect tuple streams amongst algebraic operators and their dependencies in a QEP. We assume a tree-shaped QEP generated by a conventional sequential optimizer as illustrated in Figure 2. We focus on pipelining parallelism because:

1. Pipelining parallelism is much easier to control than *independent parallelism* where operators without data dependencies are executed in parallel. Furthermore, intermediate results do not have to be fully materialized, i.e. stored in main-memory or on secondary storage, but are immediately processed by the succeeding operator.

2. For certain classes of queries, e.g. star queries, all evaluation plans are linear trees, thus pipelining parallelism is the *only* possibility to achieve parallel processing [8], at all.

Parts of the first point also apply to sequential query execution, thus, the optimizer is supposed to deliver appropriate pipelining segments. Furthermore, we restrict ourselves to processing join operators, in particular hash joins. All models presented can be adapted easily for arbitrary types of operators.

The basic unit of data transport considered is a *tuple*. The set of all tuples occurring in a query tree is denoted by D. All tuples are considered unique, regardless of their contents, i.e. an algebraic operator applied to one tuple may entails none, one, or several new tuples all distinct from the original. As a consequence, each tuple $d \in D$ is processed by exactly one operator.

Let $J = \{\bowtie_1, \ldots, \bowtie_m\}$ contain the join operators of a QEP. In pipelining processing joins can be seen as unary operators [14]: A join maps each tuple of its input to a set of output tuples, i.e. $\bowtie$: $D \to 2^D$ where 2^D denotes the power set of D. J can be ordered and thus we refer to single operators by their indexes $\{1, \ldots, m\}$. Together with tuple uniqueness a mapping $\rho : D \to \{1, \ldots, m\}$ can be defined to indicate the target operator for a tuple. The *join product* of a tuple d is the set of tuples that are entailed by d:

$$\Phi : D \to 2^D$$

with

$$\Phi(d) = \bowtie_{\rho(d)} (d)$$

These definitions enable a general treatment of the dataflow without distinction of various cases as well as relationships between operators.

Definition 2.1
A pair (E, D_0) with $E \subset J$ and $D_0 \subset D$ is called *Pipelining Segment* of length n if

$$\forall i : 1 \leq i \leq n : \quad \bowtie_i (\Phi^{i-1}(D_0)) = \Phi^i(D_0)$$

$\diamond$

D_0 is called *input relation* and $\Phi^n(D_0)$ *result relation* of (E, D_0). Moreover, $D = \cup_{i=0}^{n} \Phi^i(D_0)$ holds. The next corollary immediately follows.

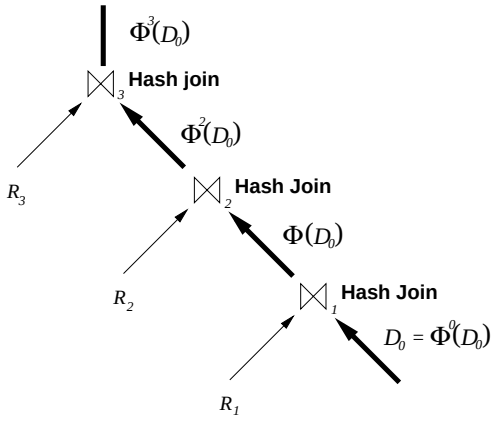

Figure 2: Pipelining Segment

Corollary 2.2
Any QEP can be decomposed into a set of pipelining segments [10].

3. Processing Model

This section describes a processing model for a the parallel processing environment on the basis of two components: a set of *queues* Q, and a set of *processing units* P. Queues implement tuple sets and a partial temporal order reflecting the arrival order. Queues are not limited to FIFO-structures, since we do not make use of the queue's particular order. The cardinality—i.e. the length—of a queue q is denoted by $|q|$. Two queues are distinguished: q_{in} which implements the initial tuple set D_0 and q_{out} that contains the result relation after the processing.

3.1 Assumptions

Our model is based on the following assumptions.

A1. Uniformity of tuple loading. Each tuples takes the same time for loading when assigned to a processing unit p_i—independently of the tuple that was assigned previously to p_i, i.e. the particular tuple arrival order is irrelevant. Tuples differ only in size of the output they entail.

A2. Exclusive processing. A tuple can only be assigned to one single processing unit. A tuple that is released by a processing unit does not re-appear at any later point of time t.

A3. Non-stop processing. Once a tuple d is assigned to a processing unit, $\Phi(d)$ is generated without any delay, break or interrupt.

A4. Homogeneity of processing units. The time to generate $\Phi(d)$ is independent of the particular processing unit, i.e. all processing units achieve the same speed.

A5. Isolation of processing units. Processing units do not affect each other, even when they consume tuples form the same queue or append to the same queue at the same time.

We further assume that (1) all hash tables of a pipelining segment fit into main memory and (2) every hash-look up gives a definite answer, i.e. no further comparison is needed. These two assumptions are not part of the model, since they are fairly common in this area of research [17, 13, 3, 10]. In Section 5, we assess these points experimentally.

3.2 Definitions

All relationships between D, Q and P are temporal. To indicate this, we use a time variable index, e.g. $a \in_t A$ means that a is element of set A at t, while $|q|_t$ denotes the length of queue q at t. For convenience, we write $a \in_{[t_1,t_2]} A$ instead of $a \in_t A$, $\forall t \in [t_1, t_2]$.

DEFINITION 3.1

Let (E, D_0) be a pipelining segment of length n, and P and Q as defined above. A *Pipelining Processing Strategy (PPS)* is a mapping

$$\mu : D \times T \to Q \cup P \cup \{\perp\}$$

with $T = [t_0, t^{(\mu)}_{stop}]$ and

1. $\forall d \in D_0 : d \in_{t_0} q_{in}$,
2. $\forall d \in \Phi^n(D_0) : d \in_{t^{(\mu)}_{stop}} q_{out}$,
3. $\forall d \in D : \exists t_1, t_2, t_3 \in T : \mu_{[t_1,t_2]}(d) \in Q \ \wedge \ \mu_{[t_2,t_3]}(d) \in P$,
4. $\mu_{[t_1,t_2]}(d) \in P \wedge \mu_{[t'_1,t'_2]}(d') \in P,\ d' \in \Phi(d) \quad \Rightarrow \quad t_1 < t'_1$
5. $\mu_t(d_1) = p \ \wedge \ \mu_t(d_2) = p \quad \Rightarrow \quad d_1 = d_2$

where $\mu_t(d) = \perp$ means, the location of d is not defined at time t. The length of T is called *execution time* of μ. ◇

The first two constraints demand, that all tuples of the initial relation D_0 are in the input queue before the processing starts, and all tuple of the result relation are elements of the output queue after the processing terminates. The third point requests the existence of an interval of time for each tuple, when it is to be processed—consisting of an interval $[t_1, t_2]$ it is stored in one of the queues and an interval $[t_2, t_3]$ it is located on a processing unit. There is no delay in between these intervals, i.e. the tuple directly goes from the queue to the processing unit. Next, a tuple d' part of the join product of tuple d cannot be processed before d. Finally, a tuple is exclusively processed, as demanded by A2.

The work needed to process a tuple d, i.e. to generate $\Phi(d)$, consists of two parts: probing against the hash table and building the new result tuples. The latter includes allocation of memory or buffer space and storing the result tuple. Thus, the work for a single tuple d totals

$$w(d) = w_{probe}(d) + \sum_{d_j \in \Phi(d)} w_{build}(d_j)$$

According to A1 and A3 we can substitute $w_{probe}(d)$ by w_{probe} and $w_{build}(d_j)$ with w_{build}, yielding

$$w(d) = w_{probe} + \sum_{i=1}^{|\Phi(d)|} w_{build} = w_{probe} + |\Phi(d)| w_{build}.$$

Because of A3 and A4 work is equal to time. Thus, the time $t^{(seq)}$ a sequential system needs for processing a particular pipelining segment is

$$t^{(seq)} = w(D) = \sum_{d \in D} w(d).$$

The *load* of a processing unit p at time t is expressed by

$$l_t^{(\mu)}(p) = \begin{cases} 1 & \text{if } \exists d \in D : \mu_t(d) = p \\ 0 & \text{else} \end{cases}$$

If no ambiguity appears, the index μ is omitted in the following.

For a certain PPS μ, producer-consumer relationships between processing units may lead to idle times of some processing units:

EXAMPLE 3.2
Consider a pipelining segment with only two join operators ($J = \{\bowtie_1, \bowtie_2\}$) and two processing units ($P = \{p_1, p_2\}$). Furthermore, let p_1 process $\bowtie_1$ and p_2 process $\bowtie_2$, respectively. p_2—processing only the output of p_1—stays idle until p_1 produced its first output tuple. However, p_1 may finish its work before p_2 does. •

For each processing unit p_i there exist $t_{su}^{(\mu)}(i)$ and $t_{sd}^{(\mu)}(i)$ such that

$$t \in [t_0, t_{su}^{(\mu)}(i)] \quad \Rightarrow \quad l_t^{(\mu)}(p_i) = 0.$$

and

$$t \in [t_{sd}^{(\mu)}(i), t_{stop}^{(\mu)}] \quad \Rightarrow \quad l_t^{(\mu)}(p_i) = 0.$$

Again, the index μ is omitted if no ambiguity occurs.

DEFINITION 3.3
For a PPS μ with D, T, P and Q as defined above the interval $[t_0, t_{su}^{(\mu)}]$ with

$$t_{su}^{(\mu)} = \max_i \{t_{su}^{(\mu)}(i)\}$$

is called *startup delay* of μ. Analogous, the interval $[t_{sd}^{(\mu)}, t_{stop}^{(\mu)}]$ with

$$t_{sd}^{(\mu)} = \min_i \{t_{sd}^{(\mu)}(i)\}$$

is called *shutdown delay*. ◇

To describe and analyze different PPSs, we need a more practical form of μ: $\Pi_t : P \times T \to D \cup \{\perp\}$ where $\Pi_t(p_i)$ indicates which tuple is processed by p_i at time t. $\Pi_t(p_i) = \perp$ if no tuple is assigned to p_i at this point of time. $\Pi_t^{(in)} : P \times T \to Q \cup \{\perp\}$ maps to the queue where $d = \Pi_t(p_i)$ was stored last and $\Pi^{(out)} : P \times T \to Q \cup \{\perp\}$ indicates the queue where all the output tuples $\Phi(\Pi(p_i))$ are appended to. Both $\Pi_t^{(in)}$ and $\Pi_t^{(out)}$ map to $\perp$ if no tuple is assigned to the processing unit.

EXAMPLE 3.4
Consider a situation, where one single tuple d is in the system and two processing units with $l_t(p_i) = 0$ are available. Let $\Phi(d) = \emptyset$, i.e. only the probing phase has to be done for this tuple. Obviously, the tuple can be assigned only to one of the processing units and as no further tuples are in any queue or are produced, no work can be redistributed. One processing unit is idle all the time. However, there is no way doing better than this. •

As the Example shows, shutdown delay cannot be avoided, in general. It rather depends on the particular data. Furthermore, a strategy can only be optimal if all work entailed per tuple can be assessed *a priori*, precisely. This knowledge is not available beforehand but only a posteriori. Without this look-ahead a strategy can only schedule the currently available units of work to the best of its abilities. This leads to our notion of optimality, called π-optimality.

Definition 3.5
A PPS μ is π-*optimal* in an interval T iff for all $[t_1, t_2] \subseteq T$

$$\sum_i l_{[t_1,t_2]}(p_i) < |P| \quad \Rightarrow \quad \int_{t_1}^{t_2} \left(\sum_{q \in Q \setminus \{q_{out}\}} |q|_t \right) dt = 0 \tag{3.1}$$

◇

The left expression holds if during the entire interval at least one processing unit is idle—not necessarily the same one all the time. The right expression allows no interval of length greater than zero during which a tuple is element of a queue, otherwise the integral cannot equal 0.

Proposition 3.6
Let μ be a π-optimal PPS and (E, D_0) a pipelining segment. The execution time $t^{(\mu)}$ is bound by

$$t^{(\mu)} = 2 - \frac{1}{r} t^{(opt)}$$

where r denotes the number of available processing units. Both $t^{(opt)}$ and $t^{(\mu)}$ are bound by $t^{(seq)}$.

P r o o f :
Let w denote the total amount of work $w = w(D)$. $\lambda \in [0, 1]$ is the ratio of sequential work, i.e. λw can only be processed sequentially due to data dependencies. The optimal processing time on r processing units is

$$t^{(opt)} = \begin{cases} \lambda w & \text{if } \lambda > \frac{1}{r} \\ \frac{w}{r} & \text{else} \end{cases}$$

For $t^{(\mu)}$ the following holds:

$$t^{(\mu)} \leq \frac{w - \lambda w}{r} + \lambda w$$

The ratio of $t^{(\mu)}$ to $t^{(opt)}$ in dependency of λ is

$$W(\lambda) = \frac{t^{(\mu)}}{t^{(opt)}} \leq \begin{cases} \frac{1}{r\lambda} - \frac{1}{r} + 1 & \text{if } \lambda > \frac{1}{r} \\ r\lambda - \lambda + 1 & \text{else} \end{cases}$$

with

$$\lim_{\lambda \leq \frac{1}{r} \to} \frac{1}{r\lambda} - \frac{1}{r} + 1 = 2 - \frac{1}{r}$$

monotonous increasing and

$$\lim_{\lambda \overset{>}{\to} \frac{1}{r}} r\lambda - \lambda + 1 = 2 - \frac{1}{r}$$

monotonous increasing as well. Thus, W is both continuous and maximal in $\lambda = \frac{1}{r}$. Leading to

$$W(\lambda) \leq 2 - \frac{1}{r}$$

To proof the general upper bound we consider the case $\lambda = 1$. No parallelism can be exploited and the upper bound for $t^{(\mu)}$ as well as for $t^{(opt)}$ is $t^{(seq)}$. □

4. Pipeline Processing Strategies

This section is devoted to the description and analysis of different parallelization strategies.

4.1 Conventional Pipeline Execution

In conventional *Pipeline Execution PE*, groups of processing units are assigned to single join operators statically [8]. So, each processing unit can process tuples for one particular join operator only (cf. Expl. 3.2).

We distinguish the two cases where either the number of joins n equals the number of processing units r or n exceeds r.

Case 1: $n = r$

Every processing unit assigned to exactly one join consumes tuples from an exclusive input queue and appends its entire output to the input queue of the subsequent processing unit. Only in case there is no successor, the output is appended to the output queue, i.e. $Q = \{q_{in} = q_1, \ldots, q_{n+1} = q_{out}\}$, $\Pi^{(in)}(p_i) = \{q_i, \perp\}$, and $\Pi^{(out)}(p_i) = \{q_{i+1}, \perp\}$. For this case, PE is specified by

$$\exists t, i : l_t(p_i) = 0 \ \wedge \ |q_i| > 0 \ \Rightarrow \ \Pi_\Delta(p_i) = d \ \wedge \ \Pi_\Delta^{(in)}(p_i) = q_i \ \wedge \ \Pi_\Delta^{(out)}(p_i) = q_{i+1} \tag{I.1}$$

with $\Delta := [t, t + w(d)]$.

Case 2: $n < r$

In this case, processing units are grouped and every group is assigned to one join. We address processing units as p_{ij} where i denotes the group and j is the index within the group. r_i is the number of processing units in group i with $r_i \geq 1$ for all i. Queues are defined as before, i.e. $Q = \{q_{in} = q_1, \ldots, q_{n+1} = q_{out}\}$, $\Pi^{(in)}(p_{ij}) = \{q_i, \perp\}$, and $\Pi^{(out)}(p_{ij}) = \{q_{i+1}, \perp\}$.

For the choice of $(r_1, \ldots, r_n) \in \mathbb{N}^n$—i.e. how to distribute the $r - n$ surplus processing units—the load balancing has to be considered: The work ν_i of group i is given by

$$\nu_i = w(\Phi^{i-1}(D_0))$$

Optimal load balancing demands that each processor is doing the r-th part of the total work, i.e. for two groups i and k

$$\frac{\nu_i}{r_i} = \frac{\nu}{r} \quad \Leftrightarrow \quad \frac{\nu_i}{\nu_k} = \frac{r_i}{r_k}$$

with $r = \sum_i r_i$ and $\nu = \sum_i \nu_i$ holds. This load balancing problem has no integer solution, in general. Thus, only an approximation can be given. The problem of finding a vector

input: $n, r, (\nu_1, \ldots, \nu_n)$;
output: $(r_1, \ldots, r_n)$;

for $i \leftarrow 1$ **to** n **do**
  $r_i \leftarrow 1$;
od;
$r_{avail} \leftarrow r - n$;
while $r_{avail} > 0$ **do**
  find j with $\frac{\nu_j}{r_j} = \max_i \left\{ \frac{\nu_i}{r_i} \right\}$;
  $r_j \leftarrow r_j + 1$;
  $r_{avail} \leftarrow r_{avail} - 1$;
od.

Figure 3: Algorithm to assign r processing units to n joins

$(r_1, \ldots, r_n) \in \mathbb{N}^n$ with $r_i \geq 1$ and $\sum_i r_i = r$ such that

$$F = \sum_{i=1}^{n} \left| \frac{\nu_i}{r_i} - \frac{\nu}{r} \right|$$

is minimal is called *Processing unit Assignment Problem (PAP).*

To obtain a discrete solution, we first assign one single processing unit to each join and then gradually add processing units always to the currently slowest join until all processing units are distributed (see Figure 3). The overall execution time—dominated by the slowest join—is step by step reduced as far as possible.

For the remainder of this Section we assume a grouping of processing units such that F is minimal. The following invariant describes PE

$$\begin{aligned} &\exists t : (\exists i, j : l_t(p_{ij}) = 0 \ \wedge \ |q_i| > 0) \ \wedge \ (\nexists k < i, l < j : l_t(p_{kl}) = 0 \ \wedge \ |q_k| > 0) \quad \Rightarrow \\ &\Pi_\Delta(p_{ij}) = d \ \wedge \ \Pi_\Delta^{(in)}(p_{ij}) = q_i \ \wedge \ \Pi_\Delta^{(out)}(p_{ij}) = q_{i+1} \end{aligned} \tag{I.2}$$

with $\Delta := [t, t + w(d)]$. The second half of the left condition is only needed to avoid ambiguities if more than one processing unit is idle at the same time. In this case we start with the processing unit with smallest index.

Corollary 4.1
PE is not π-optimal if $F > 0$.

In this case, the performance of PE suffers from the *discretization error* [17, 16].

Proposition 4.2
Let (E, D_0) be a pipelining segment of length n. PE is π-optimal iff $n = 1$.

P r o o f :
Case 1: $n = 1$
The situation where $l_t(p_i) = 0$ for some i occurs only if $|q_{in}| = 0$. As q_{in} is the only queue in $Q \setminus \{q_{out}\}$ the proposition holds.

Case 2: $n > 1$
Consider a input relation D_0 such that $\Phi^i(D_0) \neq \emptyset$. PE has a minimum startup of

$$t_{su}^{(PE)} \geq (n-1)(w_{probe} + w_{build})$$

during which at least one processing unit is idle but $|q_{in}| > 0$. □

4.2 Data Threaded Execution
With *Data Threaded Execution (DTE)* every processing unit p_i has shared access to q_{in} and exclusive access to a local queue q_i. In contrast to PE the p_i are not only performing one but every join operator in a pipelining segment, as necessary. Every processing unit appends all the output to its local queue—unless no further processing is required—from which it preferably consumes tuples. Only in case the local queue is empty, tuples from q_{in} are consumed. DTE is described by the following two invariants:

$$\begin{aligned} &\exists t, i : (l_t(p_i) = 0 \wedge |q_i| > 0) \wedge \nexists k < i : (l_t(p_k) = 0 \wedge |q_k| > 0) \quad \Rightarrow \\ &\Pi_\Delta(p_i) = d \wedge \Pi_\Delta^{(in)}(p_i) = q_i \wedge \Pi_\Delta^{(out)}(p_i) = q' \end{aligned} \tag{I.3}$$

and

$$\begin{aligned} &\exists t, i : (l_t(p_i) = 0 \wedge |q_i| = 0) \wedge |q_{in}| > 0 \wedge \nexists k < i : (l_t(p_k) = 0 \wedge |q_k| = 0) \quad \Rightarrow \\ &\Pi_\Delta(p_i) = d \wedge \Pi_\Delta^{(in)}(p_i) = q_i \wedge \Pi_\Delta^{(out)}(p_i) = q' \end{aligned} \tag{I.4}$$

where $q' = q_i$ if $\rho(d) \leq n$ and q_{out} otherwise.

Example 4.3
Let (E, D_0) be a pipelining segment of length $n = 2$, $|P| = 4$, and $|D_0| = 3$. Furthermore, $|\Phi(d)| = 10^2, d \in D_0$ and $|\Phi(d')| = 10^2, d' \in \Phi(D_0)$, i.e. in each join, every tuple finds 10^2 partners.

Processing this pipelining segment with DTE leads to the following situation: the 3 tuples of q_{in} are assigned to 3 out of 4 processing units, say, p_1, p_2 and p_3. All further processing is performed by these three processing units because of their local queues, i.e. $|q_i|, i \in \{1, 2, 3\}$ increases to 10^2 while $|q_4| = 0$ and p_4 is idle throughout the entire processing. So, 25% of the available processing resources are wasted. •

Example 4.3 shows that even in situations that offer a high potential of parallelism, DTE may yield results of only poor quality.

Proposition 4.4
Let (E, D_0) be a pipelining segment of length n. DTE is π-optimal during $[t_0, t_{sd}^{(DTE)}]$ or if $n = 1$.

P r o o f :
For $n = 1$ see Proposition 4.2.
Let $n > 1$. As long as $|q_{in}|_t > 0$ Condition I.3 implies $l_t(p_i) = 1$. The length of q_{in} is monotonously decreasing and

$$t_{sd}^{(DTE)} = \min_t |q_{in}|_t = 0$$

Which completes the proof. □
For $t > t_{sd}^{(PE)}$ situations as described in Example 4.3 may occur.

4.3 Data Threaded Execution with Redistribution
DTE/R overcomes this drawback by redistribution of intermediate results. Every processing unit appends all its output to the input queue q_{in} unless no further processing is required, i.e. $\Pi^{(out)}(p) = \{q_{in}, q_{out}\}$, $\forall p \in P$.

The following invariant specifies DTE/R

$$\begin{aligned} &\exists t, i : l_t(p_i) = 0 \ \wedge \ |q_{in}| > 0 \ \wedge \ \nexists k < i : l_t(p_k) = 0 \quad \Rightarrow \\ &\Pi_\Delta(p_i) = d \ \wedge \ \Pi_\Delta^{(in)}(p_i) = q_i \ \wedge \ \Pi_\Delta^{(out)}(p_i) = q' \end{aligned} \tag{I.5}$$

where $q' = q_i$ if $\rho(d) \leq n$ and q_{out} otherwise.

Proposition 4.5
DTE/R is π-optimal for any pipelining segment (E, D_0).

P r o o f :
Assume

$$\sum_i l_{[t_1,t_2]}(p_i) < |P|$$

holds for an interval $[t_1, t_2]$ with $t_2 - t_1 > 0$. For all intervals $[u_i, v_i] \subset [t_1, t_2]$ where $|q_{in}|_t > 0$ with $t \in \cup_i [u_i, v_i]$

$$u_i = v_i$$

immediately follows because of Invariant I.5. Consequently,

$$\int_{t_1}^{t_2} \sum_{q \in Q \setminus \{q_{out}\}} |q|_t \ dt = \int_{\cup_i [u_i, v_i]} \sum_{q \in Q \setminus \{q_{out}\}} |q|_t \ dt + \int_{T \setminus \cup_i [u_i, v_i]} \sum_{q \in Q \setminus \{q_{out}\}} |q|_t \ dt = 0$$

since all intervals $[u_i, v_i]$ are of empty size. □

5. Quantitative Assessment

To verify the analytical results, we implemented PE, DTE, and DTE/R prototypically. All experiments were carried out on SGI PowerChallenge and Onyx shared-memory machines with 4 processors. The control-flow of each processing unit is realized by a thread. Our hash table implementation reaches a performance comparable to the Berkeley Hash/DB-package, as preliminary experiments showed.

Our first approach was to use a simple (priority) queue as input queue, as private queue for each thread in DTE, and as intermediate queue between subsequent stages in PE. To guarantee proper behavior even if several threads access a single queue concurrently, we used locking to achieve exclusive access to each queue. This strict synchronization lead to poor scaleup and speedup behavior of all strategies.

We circumvented such shortcomings by a more sophisticated queue design as follows. (1) Each queue is replicated according to the number of threads that need to access the queue. Each thread preferably uses a distinct queue to get its next input tuple from or to put its output tuples to. In case of conflicts, i.e. whenever a queue is locked (or empty, in case of gets) the thread evades to another queue. For convenience, this replication of queues as well as the access/evade strategy is transparently hidden within a single interface . (2) Similarly, each single priority queue consists of several simple queues, one for each priority. Here, inserts to a queue

name	description	value
n	number of joins	1 to 16
$\|R_i\|$	cardinality of base relations	5k to 200k
υ	range of join attribute values	$1 \leq \upsilon \leq \|R_i\|$
δ	attribute value distribution of join attributes	round-robin, uniform, normal1 (mean=$\frac{\upsilon}{2}$, deviation=$\frac{\upsilon}{10}$), normal2 (mean=$\frac{\upsilon}{2}$, deviation=$\frac{\upsilon}{5}$), exponential (mean=$\frac{\upsilon}{2}$)
af_i	augmentation factor of i-th join	$af_i = \frac{\|\Phi^i(D_0)\|}{\|\Phi^{(i-1)}(D_0)\|}$

Table 1: Query Parameters

are of course restricted to the respective priority. When getting data from the queue, the highest non-empty priority is preferred. In case of conflicts, this priority is skipped, accessing the next non-empty priority instead. Of course, this releases the strict priority order of processing tuple, but therefore enables more parallelism. (3) Last but not least, one get and one insert on a single simple queue can be performed concurrently if no conflicts can occur, i.e, if the queue is long enough. This more complex queue implementation adds some extra overhead, but pays back by avoiding unnecessary locks and enabling a higher degree of parallelism instantly.

The queries investigated are marked by the parameters given in Table 1. For each configuration, we first generate the base relations according to the query specifications, then build the hash tables sequentially and after that, execute the strategy considered. To obtain stable results we took the median of 10 runs.

5.1 Wisconsin Benchmark

The initial set of experiments deal with running two queries namely joinAselB and joinCselAselB of the Wisconsin Benchmark [6]. We implemented the selection as semi join, thus, joinAselB and joinCselAselB consist of pipelining segments of length 2 and 3, respectively.

Figures 4 and 5 depict the relative execution times $\frac{t^{(\mu)}}{t^{(DTE/R)}}$ (where μ is PE, DTE, or DTE/R) for joinAselB and joinCselAselB, respectively. In this non-skewed case, the shutdown delay of DTE is minimal, i.e. DTE is nearly π-optimal. Further, DTE/R does not suffer from the overhead of the more complex queue. PE performs significantly worse than DTE and DTE/R, mainly due to discretization error, as in both queries the first operator causes ten times as much work as the other ones.

5.2 The Average Case

The next series of experiments give an overall estimate for the average case. The base relation sizes were chosen randomly from our portfolio and one of the five distribution types was used to generate the attribute value distribution. For each query, all distributions were of the same type; the particular parameters are chosen as given in Table 1. All experiments were carried out on 4 processors.

In Figure 6, the response times for *round-robin* attribute value distribution are depicted—the values are scaled to the execution time of DTE. PE is limited by the number of processors and

therefore only values for 2, 3 and 4 joins are available. DTE and DTE/R do not differ much as only few skew situations are encountered. The execution time of PE is up to 2.2 times longer than that of DTE and DTE/R.

In Figures 7 through 10, the results for the remaining distributions—*exponential*, *uniform*, *normal2*, and *normal1*—are plotted. The savings are similar to the previous case.

Besides this overall performance comparison, we also ran experiments to measure the speedup and scaleup of the different strategies. Figure 11 shows the speedup behavior of PE, DTE, and DTE/R for a two-join-query with $af_1 = 1$ and $af_2 = 1/3$. DTE and DTE/R both provide near-linear speedup, whereas PE suffers from discretization error, obviously. Similarly, Figure 12 exemplary shows the scaleup behavior of PE, DTE, and DTE/R for a two-join-query. We increased the weight of the pipelining segment with the number of processors by increasing af_2, appropriately, while leaving $af_1 = 1$. DTE and DTE/R show a negligible performance decrease of 1% when moving from one to two processors, but then, their scaleup is constant. PE shows a significantly worse scaleup behavior. Experiments with other kinds of queries show the same tendencies for both, speedup and scaleup.

5.3 Extreme Skew

Our next experiment deals with the scenario described in Example 4.3. Figure 13 depicts the relative execution times for a two-join-query with $|D_0| = 3$ and $af_1 = af_2 = 10^2$. Using one or three processors, DTE is as fast as DTE/R, as the shutdown delay of DTE is minimal (cf. Example 3.4). Using two or four processors, DTE is 33% slower than DTE/R. With two processors, first each processes one input tuple, afterwards one processes the third input tuple, while the other one is idle, i.e. $t_{sd}^{(DTE)} \approx t_0 + \dfrac{t_{stop}^{(DTE)} - t_0}{2}$. With four processors, one processor stays idle during the whole execution, while three process one input tuple each, i.e. $t_{sd}^{(DTE)} = t_0$. PE reaches the performance of DTE when using four processors, but is worse otherwise.

This experiment confirms that DTE may be non-π-optimal on $[t_{sd}^{(DTE)}, t_{stop}^{(DTE)}]$ in case of skew. Of course, the impact of this non-π-optimality on the total execution time depends on the severity of skew. To examine this impact, we used a two-join-query with base relations of equal size (24k tuples). We varied the amount of skew by choosing attribute value distributions and ranges of join attributes so that:

- In the first join, h input tuples hit and find $f = \dfrac{24000}{h}$ tuples each. Between two subsequent hitting tuples $f - 1$ tuples find no partner, i.e. every f-th input tuple finds f partners. For instance, with $h = 2$ the 12000th and the 24000th, i.e. last, tuple hit and entail 12000 output tuples each.
- Each of the other joins produces exactly one output tuple from each input tuple.

The parameter h provides a kind of metric for the amount of skew: the smaller h is, the greater the skew is. In our experiments, we varied h from 1 to 8. Figures 14 and 15 show the relative execution times for queries consisting of two joins executed on two and three processors, respectively.

With $h = 1$, DTE provides the worst performance of the three strategies. First, all processors are involved in executing the first join, i.e. just probing the input tuples against the first hash table without finding any partner. Only the last input tuple finds partners, which then have to be process through the second join by one thread, only. PE performs better than DTE, because processors assigned to the second join start working as soon as the first output tuple of the first join is produced. Here, processing tuples through the second join partly overlaps with

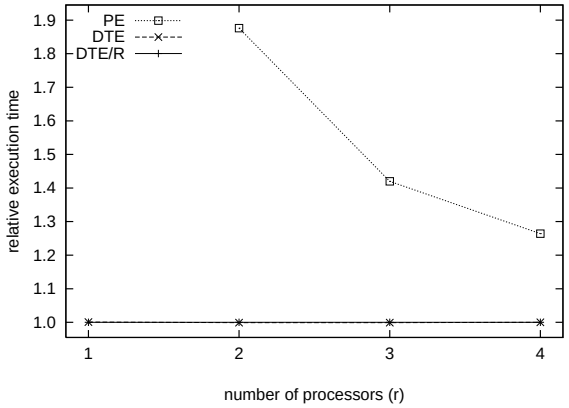

Figure 4: Wisconsin's joinAselB-query

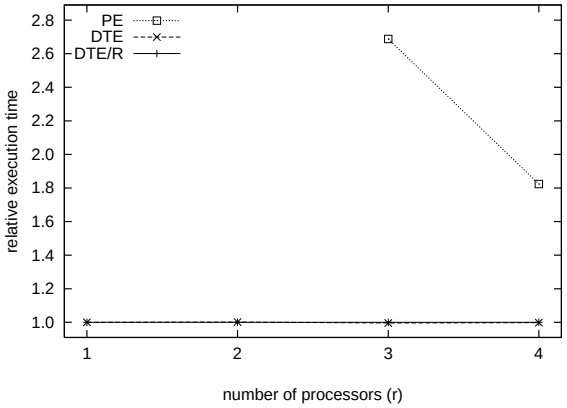

Figure 5: Wisconsin's joinCselAselB-query

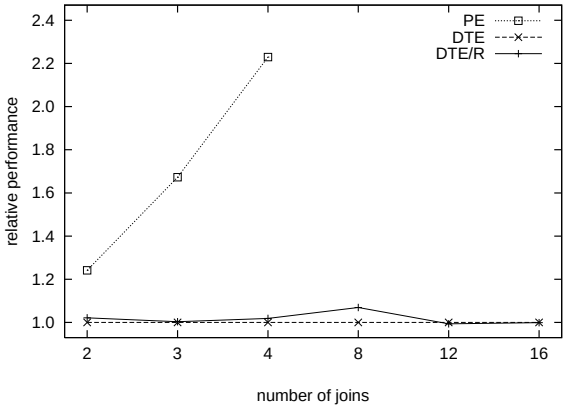

Figure 6: Average case (round-robin)

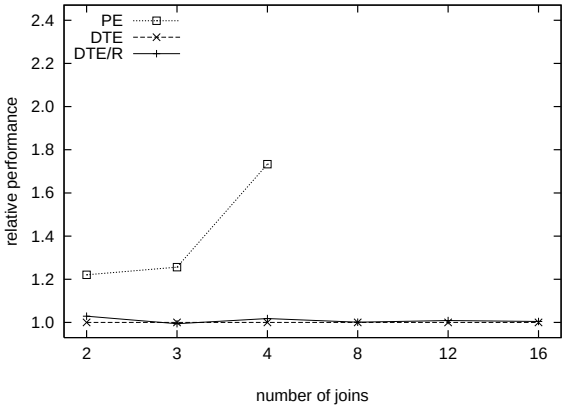

Figure 7: Average case (exponential)

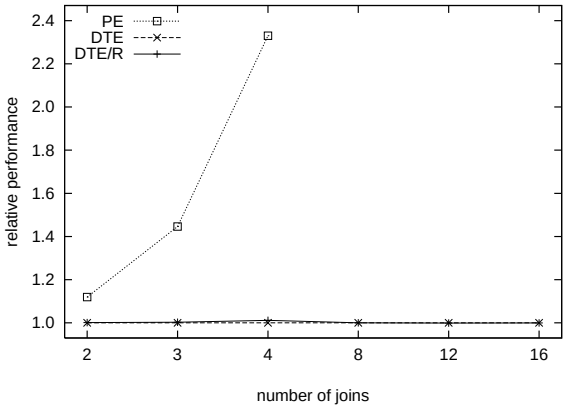

Figure 8: Average case (uniform)

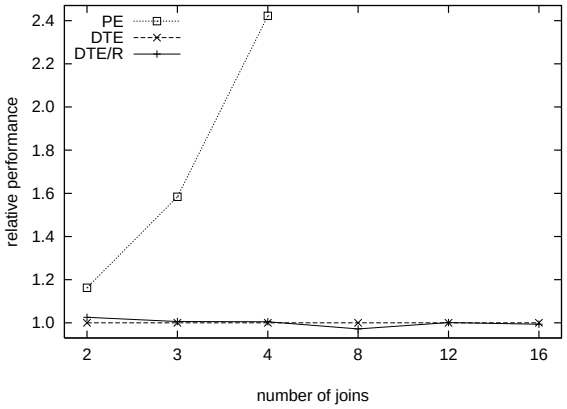

Figure 9: Average case (normal2)

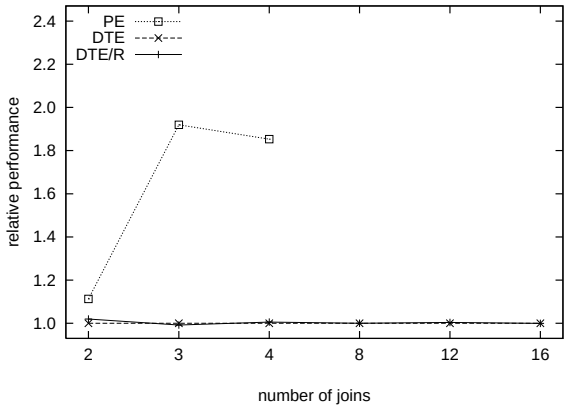

Figure 10: Average case (normal1)

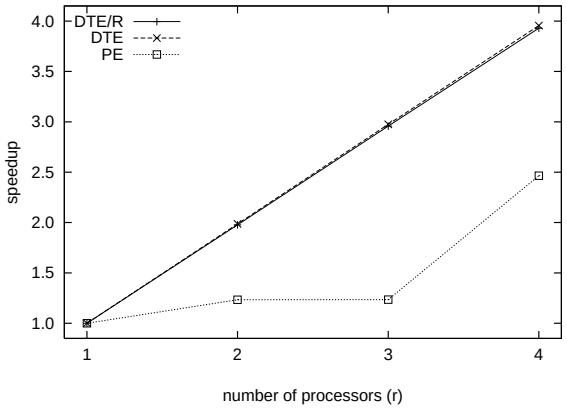

Figure 11: Speedup

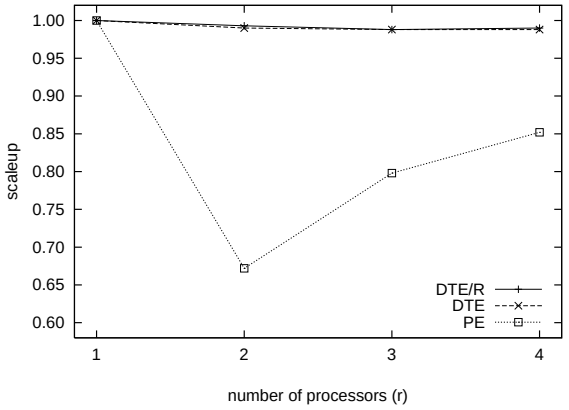

Figure 12: Scaleup

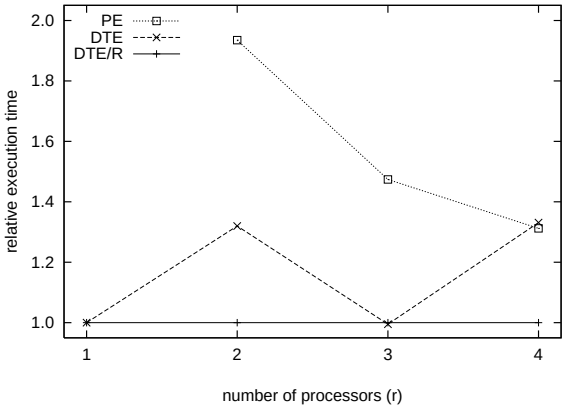

Figure 13: Query of Example 4.3

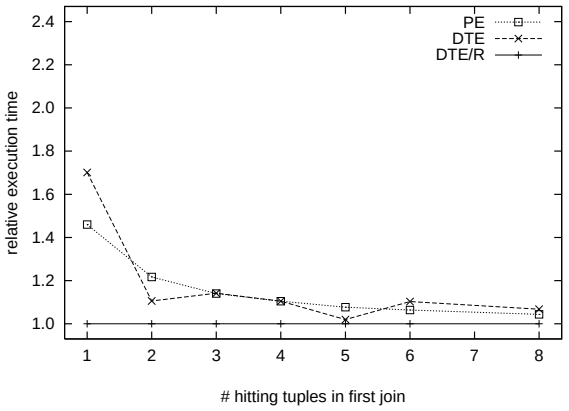

Figure 14: Skew on 2 processors

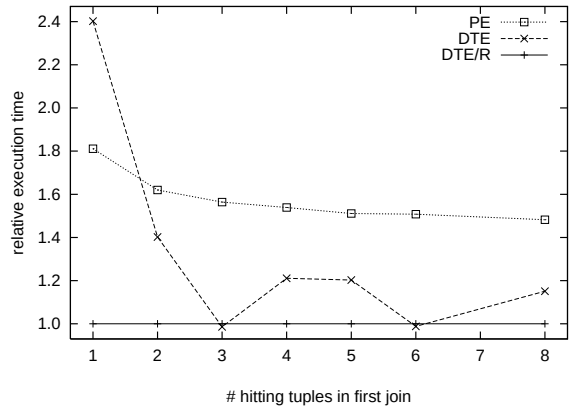

Figure 15: Skew on 3 processors

producing output tuples of the first join. DTE/R performs better than PE as already the first join is executed on twice as many processors as with PE.

With increasing h, i.e. the decreasing skew, the differences between the strategies become smaller as PE and DTE achieve better load balancing, then. For each $h > 1$, DTE is at least as fast as PE. With two processors, DTE performs similar to PE, but even reaches DTE/R, whenever h is even, i.e. a multiple of 2. With three processors, the performance of DTE is significantly better than that of PE, and reaches DTE/R, whenever h is a multiple of 3. Moving from two to three processors, the difference between PE and DTE/R increases due to discretization error.

6. Conclusion

In this paper we presented an analytical framework to improve optimization of pipelined query processing. The key notion is the concept of π-optimality, which reflects the execution strategy's ability of ad-hoc resource utilization. This notion leads to a transparent interface between the optimizer and the execution engine. Once the optimizer has determined the degree of inherent parallelism for a query plan, a π-optimal strategy ensures the best possible execution.

The approach taken has been verified by comparing the standard execution techniques with a π-optimal variant of DTE, a data parallel algorithm. Our quantitative assessment shows that π-optimal strategies are resistant against various kinds of skew, contrary to their non π-optimal counterparts.

The promising results obtained raise the question if the pre-processing phase inherently assumed in other pipelined query optimization strategies can also be efficiently integrated with π-optimal strategies. In addition, we intend to address the other main optimization issues: degree of parallelism and length of pipelining segments. We expect that our approach provides new opportunities to arrive at a powerful extensible query optimizer.

References

1. L. Bouganim, D. Florescu, and P. Valduriez. Dynamic Load Balancing in Hierarchical Parallel Database Systems. Technical Report 2815, INRIA, March 1996.
2. C. Chekuri, W. Hasan, and R. Motwani. Scheduling Problems in Parallel Query Optimization. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, San Jose, CA, USA, May 1995.
3. M.-S. Chen, M. Lo, P. S. Yu, and H. C. Young. Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins. In *Proc. Int'l. Conf. on Very Large Data Bases*, Vancouver, BC, Canada, August 1992.
4. M. N. Garofalakis and Y. E. Ioannidis. Multi-dimensional Resource Scheduling for Parallel Queries. In *Proc. ACM SIGMOD Int'l. Conf.*, Montreal, Canada, June 1996.
5. M. N. Garofalakis and Y. E. Ioannidis. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. In *Proc. Int'l. Conf. on Very Large Data Bases*, Athens, Greece, September 1997.
6. J. Gray, editor. *The Benchmark Handbook*. Morgan Kaufmann, San Mateo, CA, USA, 1993.
7. W. Hasan, D. Florescu, and P. Valduriez. Open Issues in Parallel Query Optimization. *ACM SIGMOD Record*, 25(3), September 1996.
8. W. Hasan and R. Motwani. Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelining Parallelism. In *Proc. Int'l. Conf. on Very Large Data Bases*, Santiago, Chile, September 1994.
9. W. Hasan and R. Motwani. Coloring away communication in parallel query optimization. In *Proc. Int'l. Conf. on Very Large Data Bases*, Zurich, Switzerland, September 1995.
10. H.-I. Hisao, M.-S. Chen, and P. S. Yu. On Parallel Execution of Multiple Pipelined Hash Joins. In *Proc. ACM SIGMOD Int'l. Conf.*, Minneapolis, MN, USA, May 1994.
11. W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *Proc. Int'l. Conf. on Parallel and Distr. Inf. Sys.*, Miami Beach, FL, USA, December 1991.
12. Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *Proc. ACM SIGMOD Int'l. Conf.*, Denver, CO, USA, May 1991.

13. M. Lo, M.-S. Chen, C. V. Ravishankar, and P. S. Yu. On Optimal Processor Allocation to Support Pipelined Hash Joins. In *Proc. ACM SIGMOD Int'l. Conf.*, Washington, DC, USA, May 1993.

14. S. Manegold, J. K. Obermaier, and F. Waas. Load Balanced Query Evaluation in Shared-Everything Environments. In *Proc. European Conf. on Parallel Processing*, Passau, Germany, August 1997.

15. S. Manegold, J. K. Obermaier, F. Waas, and J.-C. Freytag. Data Threaded Query Evaluation in Shared-Everything Environments. Technical Report HUB-IB-58, Humboldt-Universität zu Berlin, Institut für Informatik, April 1996.

16. J. Srivastava and G. Elsesser. Optimizing Multi-Join Queries in Parallel Relational Databases. In *Proc. Int'l. Conf. on Parallel and Distr. Inf. Sys.*, San Diego, CA, USA, January 1993.

17. A. N. Wilschut, J. Flokstra, and P. M. G. Apers. Parallel Evaluation of Multi-Join Queries. In *Proc. ACM SIGMOD Int'l. Conf.*, San Jose, CA, USA, May 1995.