Load Balanced Query Evaluation in Shared-Everything Environments*

Stefan Manegold[1] Johann K. Obermaier[1] Florian Waas[2]

[1]Humboldt-Universität zu Berlin
Institut für Informatik
10099 Berlin, Germany
⟨lastname⟩@dbis.informatik.hu-berlin.de

[2]CWI, P.O.Box 94079
1090 GB Amsterdam
The Netherlands
flw@cwi.nl

Abstract. In this paper, we present *data threaded execution*, a new strategy to exploit both, *pipelining* and *intra-operator parallelism* in shared-everything environments. Data threaded execution is intuitive, straightforward to implement, but resistant against workload estimation errors and resistant against the *discretization error* of processor scheduling conventional strategies suffer from. Furthermore, data threaded execution minimizes *startup* and *shutdown execution delays*. Simulation results show that data threaded execution outperforms conventional strategies significantly due to the better utilization of parallel processing resources.

1 Introduction

Parallel processing in database systems is one of the keys to the required performance improvements of modern database applications data require a higher performance [DG92]. In general, parallelism for the evaluation of database queries is classified into three main categories: *inter-query*, *inter-operator*, and *intra-operator parallelism*. Inter-operator parallelism with no execution dependencies between operators is called *bushy parallelism*. With a producer/consumer relationship between operators, we speak of *pipelining parallelism*. Recently, the use of inter-operator parallelism has been investigated [CLYY92, SD90, SYT93, SE93, WA91]. Pipelining parallelism is of particular interest.

The major problem with the usage of pipelining parallelism are the dependencies between operators, i.e. the performance of the pipelining execution is dominated by the slowest operator. Hence, it is important to predict the workload of the operators as precisely as possible to determine the optimal degree of parallelism for each operator. There are two main sources of errors: failures in the prediction of the operators' work (*execution skew*) and the *discretization error* [SE93, WFA95], i.e. there is no discrete processors-to-operators assignment such that every operator reaches its optimal degree of parallelism. Minimizing the discretization error by using more processes than processors as a straightforward solution adds the significant overhead of process context switching.

* supported by the German Research Council under contract DFG Fr 1142/1-1.

An additional problem with the dependencies between operators are *startup* and *shutdown execution delays* [GHK92, WFA95]. Processors assigned to operators at the end of a pipeline are idle at the beginning of the computation, whereas processors assigned to operators at the begin of the pipeline are idle towards the end of the execution.

Scope of this Paper Like in [CLYY92, SD90, SYT93] we focus in this paper on the issue of load balanced execution of pipelining segments (*PS*). We assume that an optimizer has already generated a tree-shaped query plan and partitioned the plan in PSs with the following characteristics: (1) Only the last operator of each segment may be a blocking operator, all other operators are non-blocking operators. The optimizer tuned the size of each segment so that (2) all necessary tables can be loaded in main memory and (3) all processing then can be done in main memory. To achieve this, the optimizer splits a sequence of non-blocking operators into multiple segments if necessary.

All segments are evaluated one after the other according to the producer/consumer data dependencies between them. We do not consider parallel evaluation of data independent PSs, as this obtains no performance improvements [SYT93]. Evaluation of a segment proceeds in two phases: In the first phase all inner relations of joins in the segment are loaded by parallel I/O and the (hash) indices are built in parallel. In the second phase all tuples of the outer relation are piped through selections, projections, or probe phases of joins.

The contribution of this paper is *data threaded parallel execution (DTE)*, a new parallelization strategy for efficient evaluation of the second phase of PSs on a shared-everything system. DTE allocates processing threads not to operators, but to data streams. Thus, DTE subsumes intra-operator parallelism and conventional pipelining parallelism. As additional advantages it includes load balancing, is resistant against various kinds of skew and discretization error, and avoids startup and shutdown execution delays.

The remainder of the paper is organized as follows. In Section 2, we present the pipelining query execution. Data threaded query evaluation is described in Section 3. A simulation model and a comparative performance evaluation is given in Section 4. Section 5 concludes the paper.

2 Evaluation of Pipelining Segments

Scenario To show how the different execution strategies work, we chose a rather simple example here. Of course, all strategies presented are also applicable to much more complex queries, consisting of arbitrary non-blocking operators.

We model a flight-information-system. The relation Connections consists of the attributes from and to that represent airports. Each tuple (A,B) denotes that there are non-stop flights from A to B. Table 1 shows a sample instance of Connections. We ask for connections from JFK to SBA with two stop-overs, i.e. with three single non-stop flights. We call this query JFK2SBA-query. A possible query tree for this query is depicted in Figure 1. R_i are instances of Connections, I_i are intermediate results, θ_i are the selection and join predicates, respectively.

no.	from	to
1	MEM	CLE
2	PHL	SFO
3	SFO	ABQ
4	JFK	SLC
5	GNV	JFK
6	JFK	CLE
7	CLE	MEM
8	SLC	JFK
9	ABQ	JFK
10	JFK	ABQ
11	ABQ	SFO
12	JFK	PHL
13	SFO	SBA
14	MEM	JFK
15	PHL	JFK
16	CLE	SFO
17	SBA	SFO
18	JFK	GNV
19	SFO	MEM
20	SFO	PHL

Table 1. Connections

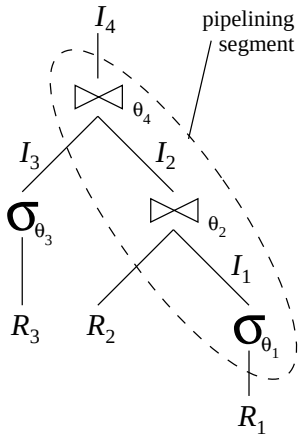

Fig. 1. Query tree

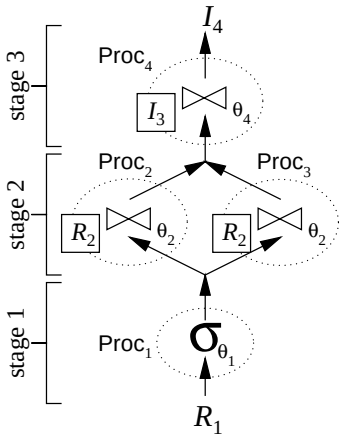

Fig. 2. PE

Pipelining Execution Model (PE) In PE each operator forms its own processing *stage*. Inter-operator parallelism is achieved by executing each stage on its own distinct set of processors. When using more processors than stages to evaluate a PS, intra-operator parallelism within stages becomes possible. The first problem to solve, is to determine the optimal degree of parallelism within each stage. Let p be the number of processors and w the work, i.e. the total sequential processing time, of the whole segment, then the optimal parallel execution time is w/p. Let n be the number of stages and w_i the work of stage i ($w = \sum_{i=1}^{n} w_i$). To achieve the optimal parallel execution time, we have to assign $\overline{p_i}$ processors to each stage i such that (1) $\sum_{i=1}^{n} \overline{p_i} = p$, (2) the parallel execution time of the stage is not longer than that of the whole segment (i.e. $w_i/\overline{p_i} \leq w/p$), and (3) the processors in the stage are never idle (i.e. $w_i/\overline{p_i} \geq w/p$). We call this *processor allocation problem (PAP)*. The resulting equation system

$$w_i/\overline{p_i} = w/p \quad \Leftrightarrow \quad \overline{p_i} = (w_i/w) \cdot p, \qquad i \in \{1, \ldots, n\}$$

does not have integer solutions for $\overline{p_i}$, in general. An algorithm to find integer approximations $p_i \geq 1$, which provide minimal response time and which fulfill $\sum_{i=1}^{n} p_i = p$. is described in the extended paper [MOW96].

A first-come-first-served policy is used to distribute the tuples that are to be processed by one stage to the processors that participate in executing this stage. Data partitioning is not necessary. Each join needs only one shared hash table, as all accesses during the probe phase are read-only, i.e. there are no conflicts. Figure 2 depicts the pipelining execution of the JFK2SBA-query on 4 processors. The shared hash tables are represented by boxes attached to the joins.

Table 2 shows a sample schedule for the JFK2SBA-query executed with PE on 4 processors. Each row represents one unit of time, called *tick*. We assume that performing a selection (S) or the probe phase of a join (P) takes one tick, whereas performing the build phase of a join (B) takes 3 ticks. Tuples of R_1

tick	stage 1 (σ_{θ_1}) processor 1	stage 2 ($\bowtie_{\theta_2}$) processor 2	stage 2 ($\bowtie_{\theta_2}$) processor 3	stage 3 ($\bowtie_{\theta_4}$) processor 4
1	$S_{\theta_1}(1)$ ∘	ⓐ	ⓐ	ⓐ
2	$S_{\theta_1}(2)$ ∘			
3	$S_{\theta_1}(3)$ ∘			
4	$S_{\theta_1}(4) \rightarrow 4.1$ •			
5	$S_{\theta_1}(5)$ ∘	$P_{\theta_2}(4.1, R_2)$		
6	$S_{\theta_1}(6) \rightarrow 6.1$ •	$B(4.1 \bowtie R_2.8)$:		
7	$S_{\theta_1}(7)$ ∘	| (JFK,SLC,JFK)	$P_{\theta_2}(6.1, R_2)$	
8	$S_{\theta_1}(8)$ ∘	$\rightarrow$ 4.1.1 •	$B(6.1 \bowtie R_2.7)$:	
9	$S_{\theta_1}(9)$ ∘	ⓑ	| (JFK,CLE,MEM)	$P_{\theta_4}(4.1.1, I_3)$ ∘
10	$S_{\theta_1}(10) \rightarrow 10.1$ •		$\rightarrow$ 6.1.1 •	ⓑ
11	$S_{\theta_1}(11)$ ∘	$P_{\theta_2}(10.1, R_2)$	$B(6.1 \bowtie R_2.16)$:	$P_{\theta_4}(6.1.1, I_3)$ ∘
12	$S_{\theta_1}(12) \rightarrow 12.1$ •	$B(10.1 \bowtie R_2.9)$:	| (JFK,CLE,SFO)	ⓑ
13	$S_{\theta_1}(13)$ ∘	| (JFK,ABQ,JFK)	$\rightarrow$ 6.1.2 •	
14	$S_{\theta_1}(14)$ ∘	$\rightarrow$ 10.1.1 •	$P_{\theta_2}(12.1, R_2)$	$P_{\theta_4}(6.1.2, I_3)$
15	$S_{\theta_1}(15)$ ∘	$B(10.1 \bowtie R_2.11)$:	$B(12.1 \bowtie R_2.2)$:	$B(6.1.2 \bowtie I_3.1)$:
16	$S_{\theta_1}(16)$ ∘	| (JFK,ABQ,SFO)	| (JFK,PHL,SFO)	| (JFK,CLE,SFO,SBA)
17	$S_{\theta_1}(17)$ ∘	$\rightarrow$ 10.1.2 •	$\rightarrow$ 12.1.1 •	$\rightarrow$ 6.1.2.1 ⊙
18	$S_{\theta_1}(18) \rightarrow 18.1$ •	ⓑ	$B(12.1 \bowtie R_2.15)$:	$P_{\theta_4}(10.1.1, I_3)$ ∘
19	$S_{\theta_1}(19)$ ∘	$P_{\theta_2}(18.1, R_2)$	| (JFK,PHL,JFK)	$P_{\theta_4}(10.1.2, I_3)$
20	$S_{\theta_1}(20)$ ∘	$B(18.1 \bowtie R_2.5)$:	$\rightarrow$ 12.1.2 •	$B(10.1.2 \bowtie I_3.1)$:
21		| (JFK,GNV,JFK)		| (JFK,ABQ,SFO,SBA)
22		$\rightarrow$ 18.1.1 •		$\rightarrow$ 10.1.2.1 ⊙
23				$P_{\theta_4}(12.1.1, I_3)$
24	ⓒ		ⓒ	$B(12.1.1 \bowtie I_3.1)$:
25		ⓒ		| (JFK,PHL,SFO,SBA)
26				$\rightarrow$ 12.1.1.1 ⊙
27				$P_{\theta_4}(12.1.2, I_3)$ ∘
28				$P_{\theta_4}(18.1.1, I_3)$ ∘

Table 2. Sample schedule (PE)

are identified by their number (cf. Tab. 1). To demonstrate how PE works, we describe the processing of one tuple throughout the PS: In stage 1, tuple 10 (JFK,ABQ) satisfies θ_1, and entails ($\rightarrow$) tuple 10.1 (JFK,ABQ). This is forwarded ($\bullet$) to stage 2, where it finds two join partners from R_2 ((ABQ,JFK), (ABQ,SFO)). Hence, tuples 10.1.1 (JFK,ABQ,JFK) and 10.1.2 (JFK,ABQ,SFO) are built and forwarded to stage 3. Tuple 10.1.1 has no join partner in I_3, thus its processing is cancelled ($\circ$). Tuple 10.1.2 finds (SFO,SBA) as join partner in I_3, so that the final output tuple 10.1.2.1 (JFK,ABQ,SFO,SBA) is built ($\odot$).

In our example, we have $(w_1, w_2, w_3) = (20, 29, 17)$ and $w = 66$. The exact solution of the PAP is $(\overline{p_1}, \overline{p_2}, \overline{p_3}) = (1.21, 1.76, 1.03)$. The best approximated processor assignment is $(p_1, p_2, p_3) = (1, 2, 1)$. This results in minimal execution times of 20, 14.5, and 17 ticks for σ_{θ_1}, $\bowtie_{\theta_2}$, and $\bowtie_{\theta_4}$, respectively. Thus, the total execution time of PE using 4 processors cannot be less than 20 ticks for the whole segment. This shows, that PE cannot reach the ideal execution time of $66/4 = 16.5$ ticks due to the discretization error. But as Table 2 shows, the actual total execution time is even worse (28 ticks). This results from two other shortcomings of PE: At the beginning, processors 2, 3, and 4 are idle as they have to wait for the tuples being produced by the previous stages (*startup execution delay*, ⓐ). For the same reason, processors 2 and 4 are idle before they finish their work (ⓑ). At the end, processors 1, 2, and 3 are idle until processor 4 has finally finished (*shutdown execution delay*, ⓒ).

3 Data Threaded Execution (DTE)

The performance of PE suffers mainly from idle time. This problem is a consequence of load balancing by static assignment of processors to stages. The key idea of our approach is to assign the available processors dynamically to the data. This leads to a much more efficient resource utilization without any additional overhead. In contrast to PE, we gather all operators of a PS into one stage and assign all processors to this stage. Obviously, this *avoids* the PAP completely.

As it is not possible to perform two operators on the same tuple in parallel, we switch from operator parallelism to data parallelism. Data parallelism covers both, intra-operator and inter-operator parallelism. We create only one thread per processor to avoid context switching and scheduling overhead. Each thread is able to perform all operations within the active PS.

Evaluation of a PS proceeds as follows: The input tuples for the PS are provided in a single queue that all threads can access. Each thread takes one tuple at a time from this queue and guides it the way through all the operators of the PS by subsequently calling the procedures that implement the operators. A tuple does not leave the thread (and thus the processor) during its way through the PS, until it has been processed by the last operator or it failed to satisfy a selection or join predicate. As soon as one tuple has left a thread, this thread is able to process the next input tuple from the queue. In the case that one tuple finds more than one partner in a join (i.e. the operator produces more than one output tuple from one input tuple), the thread has to process all these tuples first, before it can proceed with the next input tuple from the queue. Figure 3 depicts the data threaded execution of the JFK2SBA-query on 4 processors.

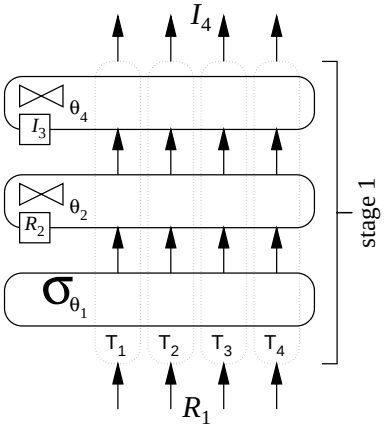

Fig. 3. DTE

Table 3 shows one possible sample schedule for the JFK2SBA-query executed with DTE on 4 processors. We use the same notation as in Table 2. There are no data dependencies between the threads, as no tuple is forwarded (•) from one thread to another. Thus, all threads start their processing simultaneously without any idle time, and none of them is idle until it finishes its work, i.e. there is no startup execution delay. A minimal shutdown execution delay (1 tick, ©) cannot be avoided. This (nearly) optimal resource utilization reduces the total execution time from 28 ticks (PE), cf. Tab. 2) to 17 ticks. Thus, in contrast to PE, DTE (nearly) reaches the minimal execution time of 16.5 ticks.

In DTE, load balancing between the processors is automatic and dynamic, as each thread can process the next input tuple as soon as it has finished the processing of the former tuple. Thus, all processors are working as long as there are input tuples in the queue. DTE optimizes resource utilization, and as no overhead is needed to achieve this, DTE minimizes the execution time.

tick	thread 1	thread 2	thread 3	thread 4
1	$S_{\theta_1}(1)$ ∘	$S_{\theta_1}(2)$ ∘	$S_{\theta_1}(3)$ ∘	$S_{\theta_1}(4) \rightarrow 4.1$
2	$S_{\theta_1}(5)$ ∘	$S_{\theta_1}(6) \rightarrow 6.1$	$S_{\theta_1}(7)$ ∘	$P_{\theta_2}(4.1, R_2)$
3	$S_{\theta_1}(8)$ ∘	$P_{\theta_2}(6.1, R_2)$	$S_{\theta_1}(9)$ ∘	$B(4.1 \bowtie R_2.8)$:
4	$S_{\theta_1}(10) \rightarrow 10.1$	$B(6.1 \bowtie R_2.7)$:	$S_{\theta_1}(11)$ ∘	(JFK,SLC,JFK)
5	$P_{\theta_2}(10.1, R_2)$	(JFK,CLE,MEM)	$S_{\theta_1}(12) \rightarrow 12.1$	$\rightarrow 4.1.1$
6	$B(10.1 \bowtie R_2.9)$:	$\rightarrow 6.1.1$	$P_{\theta_2}(12.1, R_2)$	$P_{\theta_4}(4.1.1, I_3)$ ∘
7	(JFK,ABQ,JFK)	$P_{\theta_4}(6.1.1, I_3)$ ∘	$B(12.1 \bowtie R_2.2)$:	$S_{\theta_1}(13)$ ∘
8	$\rightarrow 10.1.1$	$B(6.1 \bowtie R_2.16)$:	(JFK,PHL,SFO)	$S_{\theta_1}(14)$ ∘
9	$P_{\theta_4}(10.1.1, I_3)$ ∘	(JFK,CLE,SFO)	$\rightarrow 12.1.1$	$S_{\theta_1}(15)$ ∘
10	$B(10.1 \bowtie R_2.11)$:	$\rightarrow 6.1.2$	$P_{\theta_4}(12.1.1, I_3)$	$S_{\theta_1}(16)$ ∘
11	(JFK,ABQ,SFO)	$P_{\theta_4}(6.1.2, I_3)$	$B(12.1.1 \bowtie I_3.1)$:	$S_{\theta_1}(17)$ ∘
12	$\rightarrow 10.1.2$	$B(6.1.2 \bowtie I_3.1)$:	(JFK,PHL,SFO,SBA)	$S_{\theta_1}(18) \rightarrow 18.1$
13	$P_{\theta_4}(10.1.2, I_3)$	(JFK,CLE,SFO,SBA)	$\rightarrow 12.1.1.1$ ⊙	$P_{\theta_2}(18.1, R_2)$
14	$B(10.1.2 \bowtie I_3.1)$:	$\rightarrow 6.1.2.1$ ⊙	$B(12.1 \bowtie R_2.15)$:	$B(18.1 \bowtie R_2.5)$:
15	(JFK,ABQ,SFO,SBA)	$S_{\theta_1}(19)$ ∘	(JFK,PHL,JFK)	(JFK,GNV,JFK)
16	$\rightarrow 10.1.2.1$ ⊙	$S_{\theta_1}(20)$ ∘	$\rightarrow 12.1.2$	$\rightarrow 18.1.1$
17	ⓒ	ⓒ	$P_{\theta_4}(12.1.1.2, I_3)$ ∘	$P_{\theta_4}(18.1.1, I_3)$ ∘

Table 3. Sample schedule (DTE)

4 Quantitative Assessment

The implemented simulation framework models the structure of operators, the CPUs, the bus system, and even synchronization effects of the queuing mechanisms. As various experiments showed, our framework achieves characteristic behavior even in speed-up and scale-up.

We investigate right-deep PSs consisting of joins, only. Each join consumes materialized relations (either base relations or intermediate results) as its left input, and the results of the preceding join as its right input. Hence, queries are determined by a few parameters: The number of joins altogether, the number of tuples of the right-most input relation and the selectivities of each single join.

The *augmentation factor (AF)* denotes the ratio of the number of input tuples an operator consumes of the outer (piped) relation to the number of produced output tuples. In case of selections the augmentation factor equals the conventional selectivity. In case of joins the augmentation factor equals to $|R_I \bowtie R_O|/|R_I|$, where R_I denotes the inner relation and R_O the outer one.

Within the simulation, the respective number of output tuples produced for one single input tuple is implemented as a normal distributed number with the given AF as mean. As a consequence, the sizes of all inner relations are given implicitly and thus we do not need to model attribute values. To obtain stable results we took the arithmetic middle of at least 25 runs. The size of the right-most input relation was chosen between 10^3 and 10^5 tuples.

To examine the impact of discretization error and various kinds of data skew, separately the respective *critical* parameter is variable in each experiment, while all other parameters provide *optimal adjustments for PE.* In the final experiment, all parameters are chosen randomly to give an estimation of the average case.

The first experiment examines the impact of discretization errors. Consider a query consisting of 4 joins with an AF of 1.0 each. Whenever the number of CPUs is a multiple of 4 PE is optimal and DTE yields only poor savings ($\leq 6\%$)

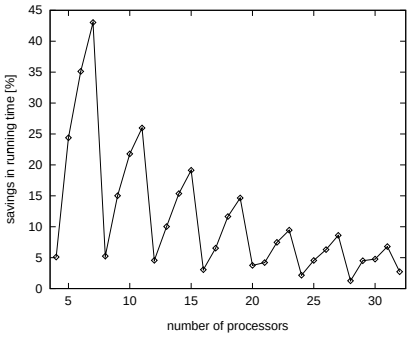

Fig. 4. Discretization errors

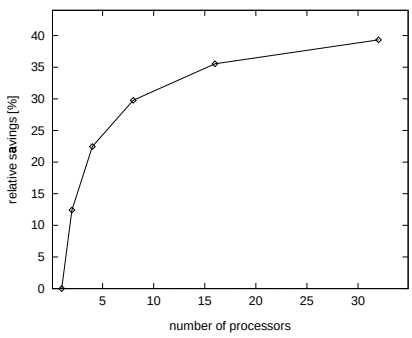

Fig. 5. Relative performance (query mix)

compared to PE that result from startup and shutdown execution delays. Contrary, in presence of discretization error DTE yields savings up to 43% (Fig. 4). Obviously, the impact of the discretization error decreases when the number of processors increases, because of the decreasing ratio of work one single processor performs to the complete work, e.g. moving from 4 to 5 processors can save 20% running time while moving from 31 to 32 can save at most 3.6%, irrespective of the query.

The following experiments examine the case where the actual execution diverges from the assumed one, the static scheduling was based on. Again, consider queries involving 4 joins with an AF of a_i for the i-th stage, where a_i is a normal distributed random number with mean 1.0. As a measure of deviation we introduce the *augmentation skew* $q = \sum_{i=1}^{n} (1 - a_i)^2$ where n is the number of stages. Our experiments show, at a skew of more than 0.35, DTE on 8 CPUs is faster than PE on 12 CPUs. For 12 and 16 CPUs, this effect already occurs at an augmentation skew of 0.225.

For a given number p of processors a query with at most p joins is generated randomly, i.e. the largest queries involve 33 base relations. The AFs are chosen randomly, too, and vary between 0.25 and 1.75. We ran 5600 different queries where each was evaluated at least 5 times with both strategies. The important observation with this experiment is that discretization error and data skew intensify each other. DTE provides savings up to nearly 40% and more than 25% at an average (Fig. 5). Note, that these results do not contradict to the previous experiments, where the amount of total work was constant and the number of processors was variable.

A detailed description of the simulation model, query configuration and further experiments can be found in [MOW96].

5 Conclusion

This paper addresses the topic of load balanced query execution in parallel database systems. We presented *data threaded execution*, a new technique for parallel query execution in shared-everything environments. Compared to con-

ventional execution methods DTE provides substantial advantages: (1) Startup and shutdown delays are minimized, (2) no discretization error arises, (3) less synchronization and inter-process communication is needed, (4) implicit load balancing establishes almost linear speedup, and (5) DTE is resistant to estimation errors during optimization.

In various simulations, we compared DTE with conventional pipelining execution (PE). In opposite to previous approaches we did not limit our considerations to idealized query parameters, but also considered configurations that cause execution and data skew. In each case, DTE outperforms the conventional pipelining execution strategies.

References

[CLYY92] M.-S. Chen, M.-L. Lo, P. S. Yu, and H. C. Young. Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 15–26, Vancouver, BC, Canada, August 1992.

[DG92] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query Optimization for Parallel Execution. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 9–18, San Diego, CA, USA, June 1992.

[MOW96] S. Manegold, J. K. Obermaier, and F. Waas. Load Balanced Query Evaluation in Shared-Everything Environments (Extended Version). Technical Report HUB-IB-70, Humboldt-Universität zu Berlin, Institut für Informatik, September 1996.

[SD90] D. A. Schneider and D. J. DeWitt. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 469–480, Brisbane, Australia, August 1990.

[SE93] J. Srivastava and G. Elsesser. Optimizing Multi-Join Queries in Parallel Relational Databases. In *Proc. of the Int'l. Conf. on Parallel and Distr. Inf. Sys.*, pages 84–92, San Diego, CA, USA, January 1993.

[SYT93] E. J. Shekita, H. C. Young, and K.-L. Tan. Multi-Join Optimization for Symmetric Multiprocessors. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 479–492, Dublin, Ireland, August 1993.

[WA91] A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proc. of the Int'l. Conf. on Parallel and Distr. Inf. Sys.*, pages 68–77, Miami Beach, FL, USA, December 1991.

[WFA95] A. N. Wilschut, J. Flokstra, and P. M. G. Apers. Parallel Evaluation of Multi-Join Queries. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 115–126, San Jose, CA, USA, May 1995.

This article was processed using the LaTeX macro package with LLNCS style