16 THINKING BIG IN A SMALL WORLD — EFFICIENT QUERY EXECUTION ON SMALL-SCALE SMPs

Stefan Manegold and Florian Waas

Centrum voor Wiskunde en Informatica, Amsterdam
{stefan.manegold, florian.waas}@cwi.nl

Abstract: Many techniques developed for parallel database systems were focused on large-scale, often prototypical, hardware platforms. Therefore, most results cannot easily be transfered to widely available workstation clusters such as multiprocessor workstations. In this paper we address the exploitation of pipelining parallelism in query processing on small multiprocessor environments. We present DTE/R, a strategy for executing pipelining segments of arbitrary length by replicating the segment's operator. Therefore, DTE/R avoids static processor-to-operator assignment of conventional processing techniques. Consequently, DTE/R achieves automatic load-balancing and skew-handling. Furthermore, DTE/R outperforms conventional pipelining execution techniques substantially.

Keywords: parallel and distributed databases, parallel query processing, dynamic load balancing.

16.1 INTRODUCTION

Parallel database research is driven by the need for higher performance demanded by new application areas of relational database technology, e.g. data warehousing. Much work has been devoted to basic research and produced notable insights to many aspects of parallel query processing. The most prominent representatives are the Gamma Database Machine on a Vax-Cluster and on an Intel Hypercube (DeWitt et al., 1986), Volcano, also running on an Intel Hypercube (Graefe, 1990), EDS/DBS3 (Bergsten et al., 1991) on an Encore MULTIMAX and on a KSR1 in a later version, and PRISMA/DB on top of a Philips/Motorola 128 processor experimental prototype (Apers et al., 1992). The first commercial database machine, Teradata's DBC/1012, was also implemented on non-standard hardware, where processors are connected via a tree-shaped network (Carino and Kostamaa, 1992).

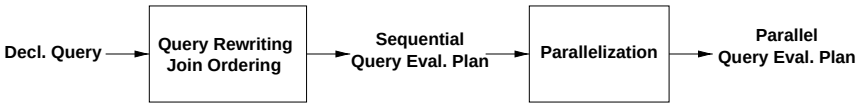


Figure 16.1 Two-phase optimization approach

An important aspect that these research prototypes have in common is that they are not affordable to the average user who just wants to reduce response time of his query execution engine. On the other hand, small SMPs (Symmetric Multi-Processors) with about 4 to 8 processors have become quite popular in recent years. These machines provide surprisingly high computing power, efficient shared memory access (fast communication) and are inexpensive, as compared to the previously mentioned research equipment.

In this paper we address the question of how to improve query execution techniques on small-scale shared-everything platforms. We present an execution strategy named DTE/R that exploits pipelining parallelism aggressively. DTE/R replicates the pipelining segment for each processor and avoids any static processor-to-operator assignments. Therefore, DTE/R circumvents the drawbacks of conventional pipelining execution, as given in (Hasan and Motwani, 1994). The basic algorithm distributes the input data over all available processors where the entire pipelining segment is processed. However, in typical skew situations, this may overload some processors while others are idle. DTE/R overcomes this problem by a redistribution of the intermediate processing results. This redistribution adds little overhead, but pays back significantly in almost every case of skew. The quantitative assessment of our implementation, carried out on a 4 processor SMP, shows substantial savings over the conventional pipelining execution technique. Furthermore, DTE/R achieves near-linear speedup and a scaleup ratio close to 1.

16.2 PARALLEL QUERY PROCESSING

Query processing consists of the two components: *query optimization* and *query execution*. The declarative query submitted by the user is transformed into a *query evaluation plan*, which is the actual program the database system has to execute. Following a common approach (Hong and Stonebraker, 1993; Chekuri et al., 1995), query optimization in parallel environments can be split into two phases: in the first step, transforming the declarative query into a procedural but sequential one, and, in the second step, parallelizing this sequential plan (Figure 16.1). This way of optimization delivers a parallel plan, consisting of the structural information of a sequential plan and an assignment of processors to sub-plans.

16.2.1 Query Optimization

We assume a conventional sequential optimizer to perform the first optimization step. In Figure 16.2, the possible result of such an optimization is depicted as a tree, consisting of hash-joins only. For convenience we focus on the processing of join trees,

although our algorithms are not restricted to a certain set of operators. Joins are considered to be the most expensive operators in relational query processing, involving disk and network access. Thus, most related work focuses on join processing as well (Schneider and DeWitt, 1990; Wilschut and Apers, 1991; Chen et al., 1992; Hong and Stonebraker, 1993; Hasan and Motwani, 1995).

A query evaluation plan can be interpreted as a dataflow graph describing *producer-consumer* dependencies between operators. The data to be processed is forwarded from one operator to the next according to the dependencies. Processing a single join involves comparing all data from one input with every data item from the second input. If both tuples fulfill the join's predicate (match on a set of attributes), a new tuple is constructed. If no matching partner is found further processing of this particular tuple is canceled. The most efficient way to perform a join is to store one input in a hash table and do a hash lookup for every tuple of the second input (for more details we refer to (Graefe, 1993)). We call an edge in the dataflow graph *blocking* if it describes input of data that has to be read entirely before any data of the second input can be processed, otherwise we call it *non-blocking* edge. In Figure 16.2, all non-blocking edges are emphasized by thick lines. Furthermore, we call operators along a chain of non-blocking edges *non-blocking operators*.

16.2.2 Exploiting Pipeline Parallelism

The data dependencies, given by the evaluation plan, suggest different feasible kinds of parallelism. *Pipelining parallelism* is of particular interest for the following reasons:

1. Pipelining parallelism is much easier to control than *independent parallelism* where operators without data dependencies are executed in parallel. The resource allocation for a *pipelining segment*—which is a series of operators along non-blocking edges—can be determined more precisely (Srivastava and Elsesser, 1993). Furthermore, intermediate results do not have to be stored in main memory or on secondary storage; they are immediately processed by the succeeding operator. Only the final output of a pipelining segment has to be stored.

2. For certain classes of queries, all evaluation plans are linear trees. Thus pipelining parallelism is the *only* possibility to achieve parallel processing (Hasan and Motwani, 1994).

3. Every evaluation plan can be decomposed into linear subtrees. Consequently, pipelining parallelism is an option for every evaluation plan.

Parts of the first point also apply to sequential query execution. Thus, we assume that the optimization step already delivers the appropriate pipelining segments. 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 (no further comparison is needed).

Table 16.1 Notation

name	*description*
n	number of joins
I_0, R_i	base relations, $i \in \{1, \ldots, n\}$
I_i	intermediate results: $I_i = R_i \bowtie I_{i-1}$, $i \in \{1, \ldots, n\}$
$\|R\|$	number of tuples in relation R
sf_i	selectivity factor of i-th join: $sf_i = \frac{\|R_i \bowtie I_{i-1}\|}{\|R_i\| \cdot \|I_{i-1}\|}$
af_i	augmentation factor of i-th join: $af_i = sf_i \cdot \|R_i\|$
w_i	weight (i.e. sequential execution time) of the i-th join
w	weight of the whole pipelining segment: $w = \sum_{i=1}^{n} w_i$
p	number of processors
p_i	number of processors assigned to the i-th stage: $p = \sum_{i=1}^{n} p_i$
$T_i(p)$	parallel execution time of the i-th stage using p processors

16.2.3 Query Execution

Once an evaluation plan is generated and decomposed into pipelining segments, the plan is executed on the target platform. Its processing demands a loading phase, where all data belonging to blocking edges is preprocessed in an appropriate way—e.g., all hash tables are built. We refer to this phase as the *build-phase*. Hereafter, the actual hash-join processing starts, called the *probe-phase*.

The build-phase is common to all execution strategies and is usually I/O-bound. Therefore, it is not interesting for purposes of this paper.

16.3 EVALUATING PIPELINING SEGMENTS

Before presenting different strategies to evaluate pipelining segments, we informally introduce some definitions and notations used in the remainder of this paper.

16.3.1 Definitions and Notations

Figure 16.2 depicts a sample pipelining segment consisting of three joins. The corresponding notation convention is given in Table 16.1.

We define the *weight* w_i of a single operator op_i as its *sequential execution time* T_i, the time a single processor needs to process all the respective input data through that operator. The weight w of a whole pipelining segment using sequential execution then amounts to the sum of the weight of all the operators within the pipelining segment: $w = \sum_{i=1}^{n} w_i$. The *ideal* parallel execution time of an operator executed on p_i processors is $T_i(p_i) = \frac{w_i}{p_i}$. Thus, the ideal execution time of the whole segment is $T = \frac{w}{p}$.

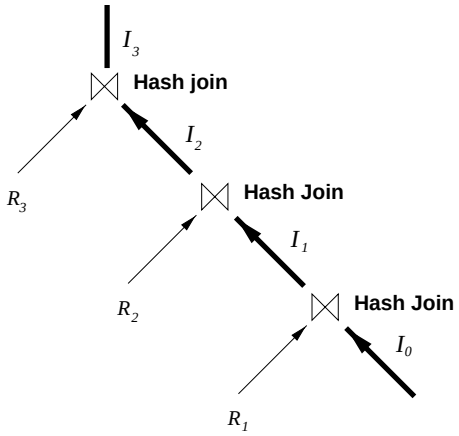


Figure 16.2 Pipelining Segment

16.3.2 Pipelining Execution

The conventional *Pipelining Execution (PE)* technique evaluates pipelining segments as follows. Each operator is executed on a distinct processor. To avoid context switching overhead, each processor executes only one operator at a time. Each output tuple produced by op_i is immediately forwarded to its consumer op_{i+1}, and op_i can then process the next input tuple.

Obviously, PE demands $n \leq p$, i.e. the number of operators has to be less than the number of processors. We distinguish two cases:

Case 1: $n = p$
Due to the data dependencies between the operators, the total execution time is dominated by the slowest operator, i.e. $T_{\text{PE}} = \max_{i=1}^{n}\{T_i\}$. If all operators have the same weight ($w_1 = w_2 = \ldots w_n = w'$), then PE provides optimal parallel performance:

$$T_{\text{PE}} = \max_{i=1}^{n}\{w_i\} = \frac{nw'}{n} = \frac{w}{p}.$$

Otherwise, PE performs suboptimally as it is not able to balance the different loads:

$$T_{\text{PE}} = \max_{i=1}^{n}\{w_i\} = \frac{n \max_{i=1}^{n}\{w_i\}}{n} > \frac{w}{p}.$$

Actually, the performance of PE is even worse. At the beginning, only one processor is active, executing the first operator. The other processors are idle and wait for the output produced by the preceding operator(s). Thus, every processor starts working with a certain *startup delay*:

$$t_{0,1} = 0\ ; \qquad t_{0,i+1} = t_{0,i} + \frac{w_i}{|I_{i-1}|}\frac{1}{af_i} = \frac{1}{|I_0|}\sum_{j=1}^{i}\frac{w_j}{\prod_{k=1}^{j} af_k}\ .$$

Similarly, as soon as the first processor has finished, the subsequent processors are still active processing the latest output of the first one. Thus, in addition to the aforementioned startup delay, there also is a *shutdown delay* that further decreases the parallel performance of PE. PE provides no means to balance the load variances at run-time.

Case 2: $n < p$

When using more processors than operators, we can assign more than one processor to the same operator. In addition to pipelining parallelism, intra-operator parallelism becomes possible—at least for some operators. To do so, the *processor allocation problem (PAP)*, to assign the appropriate number p_i^* of processors to each operator, has to be solved. We look for a configuration where

$$\frac{w_1}{p_1^*} = \cdots = \frac{w_n}{p_n^*} = \frac{w}{p}$$

holds. Unfortunately, this equation system does not have discrete solutions $(p_1^*, \ldots, p_n^*) \in \mathrm{N}^n$, in general. Thus, integer approximations $(p_1, \ldots, p_n) \in \mathrm{N}^n$ must be found, which provide minimal execution time and fulfill

$$\sum_{i=1}^{n} p_i = p.$$

In this case, the minimal achievable execution time is greater than $\frac{w}{p}$. This effect is known as *discretization error* (Srivastava and Elsesser, 1993; Wilschut et al., 1995). In (Manegold et al., 1998), we present an algorithm which computes such approximations.

16.3.3 Data Threaded Execution

With *Data Threaded Execution (DTE)* all operators of a pipelining segment are gathered into one stage that is replicated for each processor. DTE creates one thread per processor to perform all operators within the active pipelining segment. A global input queue, accessible by all threads, provides the input tuples for the pipelining segment. Each thread takes one tuple at a time and processes it through all the operators of the pipelining segment. A tuple does not leave the thread—and thus the processor—during its way through the pipelining segment, until it has successfully passed the last operator or it failed to find a join partner. Whenever a tuple leaves a thread, this thread immediately starts processing the next input tuple from the global queue, unless the queue is already empty. In case an operator produces more than one output tuple from one input tuple (a tuple finds more than one partner in a join), the originator thread has to process all these additional tuples first, before it can proceed with the next input tuple from the queue.

Contrary to PE, DTE dynamically assigns processors to the data instead of statically assigning processors to operators according to their weight. This way, DTE avoids the PAP and, hence, cannot suffer from discretization error. Further, as there are no data dependencies between the threads, DTE is not affected by startup delay.

In (Manegold et al., 1997), we presented DTE in detail and showed that in the case of non-skewed data DTE achieves optimal load balancing and, thus, minimal execution time $T_{\text{DTE}} = \frac{w}{p}$.

16.4 SKEW HANDLING

Now we will have a more detailed look at both strategies and discuss how each performs in the presence of highly skewed data. Skew in general means that due to the attribute value distributions not every input tuple finds exactly one output tuple in each join. Some of these skew situations are of little interest; for example, if—per join—only each k-th ($k > 1$) input tuple finds (exactly) one partner or (nearly) each input tuple finds (approximately) $l > 1$ partners. This just involves operators with different weights. While PE suffers from such situations due to discretization error, DTE still achieves optimal load balancing. For a detailed discussion of these kinds of *uniform* skew, we refer the interested reader to (Manegold et al., 1997). We will focus on more extreme kinds of skew in the remainder of this paper.

16.4.1 Execution Skew

PE uses cost estimates to find an appropriate assignment of processors to operators. The errors in cost estimates propagate and intensify while processing a pipelining segment. The best processor assignment according to the (erroneous) cost estimation is, apart from some rare cases, no longer the optimal assignment, according to the actual execution costs. This *execution skew* in turn leads to a non-optimal execution behavior of PE.

In contrast, DTE does not need any cost estimates for processor assignment, but rather assigns processors to the data automatically during execution. Hence, DTE does not suffer from any execution skew.

16.4.2 Join Product Skew

In the following, we assume an ideal setup for PE where neither discretization error nor execution skew occur. We will show that despite these assumptions there are several situation where PE does not perform very well. But we will also show that even DTE may perform badly in some of these situations.

Consider that case of a skewed attribute value distribution. In one of the joins each k-th input tuple ($k \gg 1$) finds k partners while all the other input tuples do not find any partners at all. This scenario, called *join product skew* (Walton et al., 1991), is typical for foreign key joins. In order to trigger just one effect at a time (in this case join product skew), we chose this scenario so that no discretization error occurs. Each join produces as many output tuples as it receives input tuples.

With PE, whenever several subsequent input tuples of an operator do not find any join partner, the next operators are idle, as they get no input data. Thus, join product skew increases the startup delay. In the same way, join product skew increases the shutdown delay, if the last input tuples processed by an operator produce lots of output tuples.

As there are no data dependencies between the threads, DTE does not suffer from startup delay at all. Thus, join product skew obviously cannot increase startup delay with DTE.

The only situation where join product skew can affect the performance of DTE arises when in one operator b input tuples ($b < p$) generate so many additional output tuples that the b threads processing these tuples are occupied significantly longer than the other $p - b$ threads need to process all the remaining input tuples from the global queue. In this case, shutdown delay also occurs with DTE: $p - b$ processors become idle as soon as the global queue is empty, while b processor are still busy. Thus, load balancing is no longer optimal.

16.4.3 DTE/R

To overcome DTE's shutdown delay, we refine the strategy in the following way. Whenever a tuple generates more than one output tuple in an operator, all but one of these output tuples are put back into the global queue. Only one tuple stays on the thread for further precessing. With this *re-distribution*, the additional output tuples can be processed by any thread that becomes idle. Thus, the refined strategy *DTE/R* achieves optimal load balancing even in the case of extreme skew.

To make this re-distribution work, in DTE/R each tuple that is put into the global queue is tagged with the id of the operator that puts it there. Original input tuples are tagged with 0. Thus, each thread knows at which operator it has to start/continue the processing of a tuple it receives from the queue.

As an extension, we changed the global queue into a preemptive queue that uses the tag as the priority of each tuple. Tuples from the queue are then processed according to their priority (highest priority first). This technique achieves two things: the queue does not become unnecessarily long and the output is not unduly deferred. For performance reasons, we implemented the preemptive queue internally as a set of simple queues (one per priority) with one common interface.

16.5 QUANTITATIVE ASSESSMENT

In order to analyze and compare the execution behavior of PE, DTE, and DTE/R, we implemented all three strategies in a prototype query engine to evaluate the probe phase of pipelining segments. We used a 4-processor SGI PowerChallenge shared-memory machine to run our experiments on. Our experiments cover arbitrary randomly generated queries as well as queries particularly designed to examine extreme skew situations. Due to space limitations, we mainly present the results for situations with extreme skew in this section. Further experiments covering non-skew, uniform skew, and execution skew situations can be found in (Manegold et al., 1998).

16.5.1 Query Design and Benching Strategy

In our experiments, queries are marked by the parameters given in Table 16.2. More details about the queries will be given for each set of experiments.

Table 16.2 QueryParameters

name	*description*	*value*
n	number of joins	1 to 16
$\|R_i\|$	cardinality of base relations	5k to 200k
r	range of join attribute values	$1 \leq r \leq \|R_i\|$
d	attribute value distribution of join attributes	round-robin, uniform, normal1 (mean=$\frac{r}{2}$, deviation=$\frac{r}{10}$), normal2 (mean=$\frac{r}{2}$, deviation=$\frac{r}{5}$), exponential (mean=$\frac{r}{2}$)

For each configuration, we first generated the base relations according to the query specifications. Then we built all the hash tables and the global input queue in main memory. We chose our query configurations so that the global input queue always fit in main memory. Thus, we could examine the pure performance of the probe phase without any I/O-influence. In case the input queue does not fit completely in main memory, the I/O to read it from disk during the probe phase affects all strategies. After that, we executed the probe phase using PE, DTE, and DTE/R. We only measured the execution time for the probe phase. We ran each distinct configuration 10 times and computed—per strategy—the median of all 10 runs as final execution time T_{PE}, T_{DTE}, and $T_{\text{DTE/R}}$, respectively.

16.5.2 The Average Case

The first series of experiments gives 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 actual attribute value distribution. For each query, all distributions were of the same type; the particular parameters were chosen as given in Table 16.2. All experiments were carried out on 4 processors.

In Figure 16.3, the response times for uniform 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. Both provide savings of up to 120% compared to PE.

In Figure 16.4, the results for the *normal2* distributed attribute values are plotted. The savings are similar to the previous case. Other distributions showed very similar results—thus we omit them here. Results can be found in (Manegold et al., 1998).

Besides this overall performance comparison, we also ran experiments to measure the speedup and scaleup of the different strategies. Figure 16.5 shows the speedup behavior of PE, DTE, and DTE/R for a two-join-query with $af_1 = 0.5$ and $af_2 = 1$. DTE and DTE/R both provide near-linear speedup, whereas PE obviously suffers from discretization error. Similarly, Figure 16.6 shows the scaleup behavior of PE,

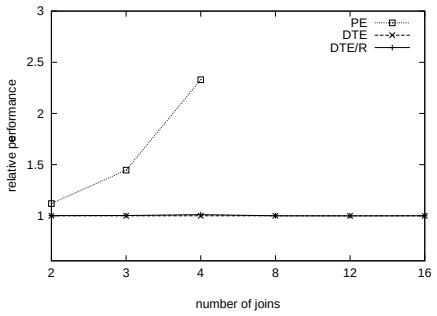


Figure 16.3 Uniform

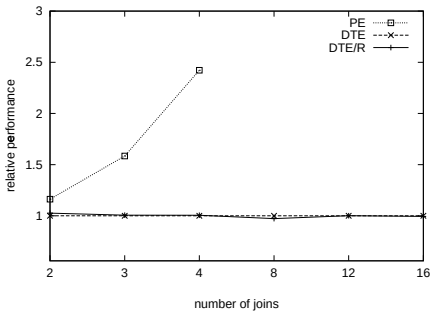


Figure 16.4 Normal2

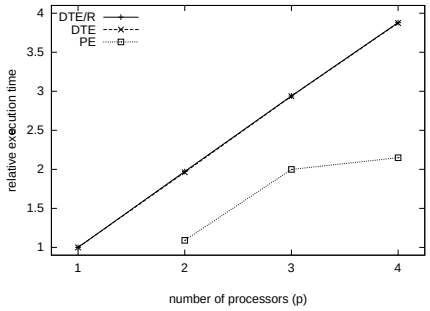


Figure 16.5 Speedup

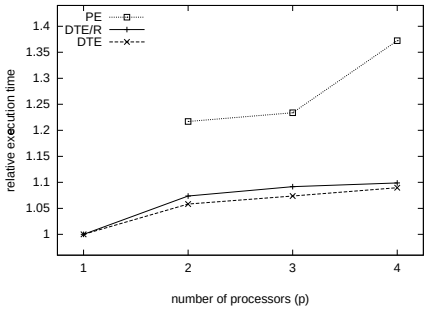


Figure 16.6 Scaleup

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_1, appropriately, while leaving $af_2 = 1$. DTE and DTE/R show a slight performance decrease of 6-7% 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.

16.5.3 Extreme Skew

As mentioned in the previous section, DTE cannot achieve optimal load balancing once $b < p$ input tuples generate too many additional output tuples. When this happens, the b threads processing these tuples are occupied significantly longer than the other $p - b$ threads need to process all the remaining input tuples from the global queue.

To examine such situations, we used a round-robin distribution on base relations of equal size (24k tuples) and chose the ranges of the join attributes so that:

- In the first join, m input tuples hit and find $k = \frac{24,000}{m}$ tuples each. Between two subsequent hitting tuples, $k-1$ tuples find no partner. In other words, every

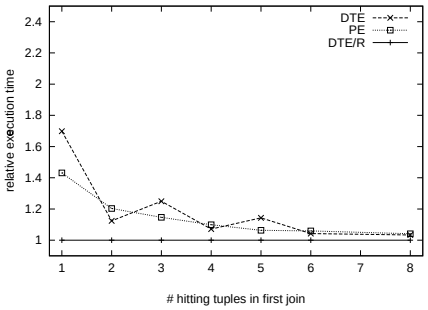


Figure 16.7 Skew on 2 processors

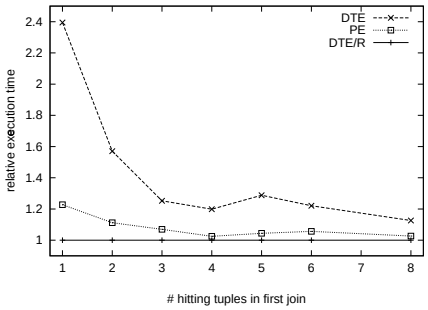


Figure 16.8 Skew on 4 processors

k-th input tuple finds k partners. For instance, with $m = 2$ the 12,000th and the 24,000th (last) tuple hit and generate 12,000 output tuples each.

- Each of the other joins produces exactly one output tuple from each input tuple.

The parameter m provides a kind of metric for the amount of skew: the smaller m is, the greater the skew is. In our experiments, we varied m from 1 to 8. Figures 16.7 and 16.8 show the relative execution times $\frac{T_s}{T_{\text{DTE/R}}}$, $s \in \{\text{PE}, \text{DTE}, \text{DTE/R}\}$ for queries consisting of two joins executed on two and four processors, respectively.

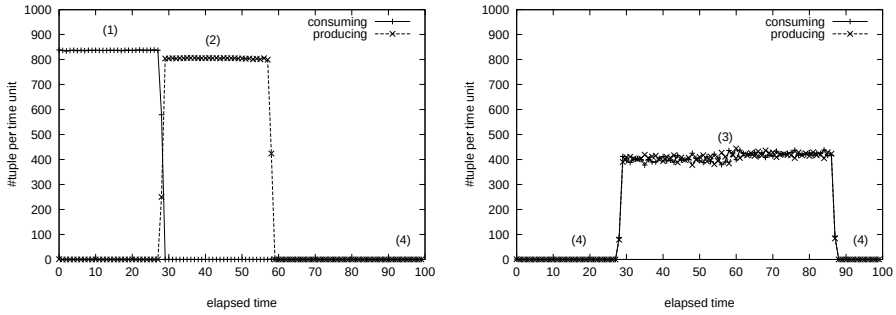
With $m = 1$, DTE provides the worst performance of the three strategies. First, all processors are involved in executing the first join, i.e. just probing the input tuples against the first hash table without finding any partner. Only the last input tuple finds partners, which then have to be processed through the second join by one only thread. PE performs better than DTE, as the 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 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.

As m increases, the skew decreases. Now the differences between the strategies becomes smaller as the load balancing is easier for PE and DTE. With two processors, DTE is better than PE whenever m is even. With four processors, the performance DTE nearly reaches that of PE whenever m is a multiple of 4, but DTE is never better than PE.

To get better insight in what happens during execution with the different strategies, we also measured the rates at which each thread of each strategy consumes and produces tuples. Figures 16.9 through 16.11 show the respective curves for PE, DTE, and DTE/R executing the skew query with $m = 1$ on two processors. The elapsed time is normalized to the slowest execution time. The different phases during execution are denoted by numbers as follows:

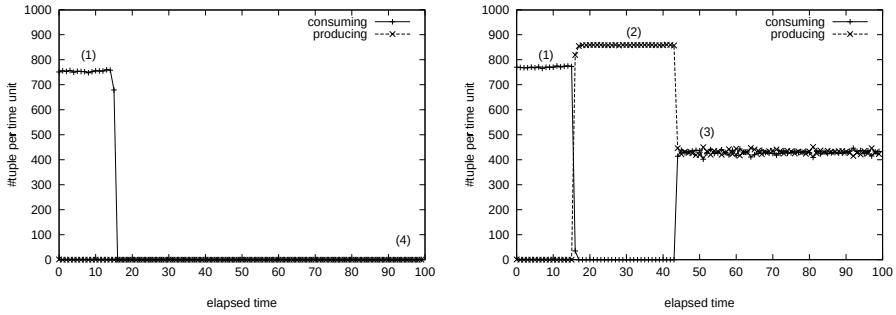
(1) consuming all tuples from the input queue and probing them in the first join without finding any join partner for all but the last tuple,

(2) producing 24k output tuples from the last input tuple of the first join,



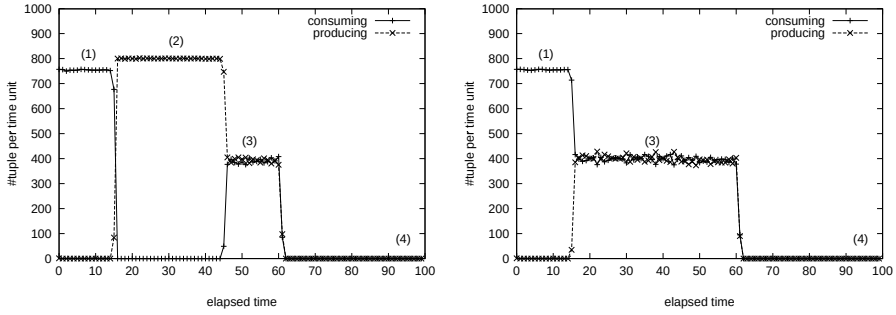
Processor 1 Processor 2

Figure 16.9 Rates of consuming and producing tuples (PE)



Thread 1 Thread 2

Figure 16.10 Rates of consuming and producing tuples (DTE)



Thread 1 Thread 2

Figure 16.11 Rates of consuming and producing tuples (DTE/R)

(3) consuming 24k tuples from the first join, probing them in the second join, and producing one output tuple from each one, and

(4) idle time.

In our implementation, both probing an input tuple against a hash table as well as producing an output tuple take approximately the same time. Thus, performing only one of these actions (1,2) can be done at a rate that is approximately twice as high as that of performing both actions (3).

Figure 16.9 shows the PE results. Only the first processor executes phase (1). As soon as it starts producing output tuples from the last input tuple (2), the second processor starts consuming these tuples and processes them through the second join (3).

Figure 16.10), shows the DTE results. Both threads participate in executing phase (1). Then, only thread 1 executes phases (2) and (3).

Figure 16.11) shows the DTE/R results. Both threads share the execution of phase (1). After that, thread 1 puts each tuple produced in phase (2) immediately into the global queue, so that thread 2 can consume these and process them through phase (3). As soon as thread 1 has finished phase (2), it joins phase (3).

For queries involving up to 16 joins, DTE and DTE/R show the same tendencies as for short queries.

16.6 CONCLUSION

In this paper, we addressed the problem of utilizing the computing power of small off-the-shelf SMP workstations in query execution. We presented the key ideas of DTE/R, a novel strategy to exploit pipelining parallelism in query processing. The principles of operator replication and data threading are building blocks for the desired improvement.

In (Manegold et al., 1997) we reported on DTE, the basic algorithm and gave performance assessments for situations not involving skew. However, in extreme skew situations DTE's performance varies enormously. Here, we show that the potential problems resulting from skew can be solved by a re-distribution mechanism, which adds some extra overhead but makes the algorithm resistant to any skew.

The final contribution of this paper is the assessment of a real implementation, not merely a simulation. We experimented with a broad class of queries and examined both average case and extreme situations. The main characteristics of DTE/R are a nearly linear speedup and a scaleup ratio close to 1. DTE/R outperforms conventional pipelining technique substantially.

Our experiments are promising and show that highly efficient parallel database techniques can be beneficial for small parallel configurations.

References

Apers, P. M. G., van den Berg, C. A., Flokstra, J., Grefen, P. W. P. J., Kersten, M. L., and Wilschut, A. N. (1992). PRISMA/DB: A Parallel Main Memory Relational DBMS. *IEEE Trans. on Knowledge and Data Eng.*, 4(6):541–554.

Bergsten, B., Couprie, M., and Valduriez, P. (1991). Prototyping DBS3, A Shared-Memory Parallel Database System. In *Proc. Int'l. Conf. on Parallel and Distr. Inf. Sys.*, pages 226–235, Miami Beach, FL, USA.

Carino, F. and Kostamaa, P. (1992). Exegesis of DBC/1012 and P-90 – Industrial Supercomputer Database Machines. In *Proc. Int'l. Conf. on Parallel Architectures and Languages in Europe*, pages 877–892.

Chekuri, C., Hasan, W., and Motwani, R. (1995). Scheduling Problems in Parallel Query Optimization. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 255–265, San Jose, CA, USA.

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

DeWitt, D. J., Gerber, R. H., Graefe, G., Heytens, M. L., Kumar, K. B., and Muralikrishna, M. (1986). GAMMA — A High Performance Dataflow Database Machine. In *Proc. Int'l. Conf. on Very Large Data Bases*, pages 228–237, Kyoto, Japan.

Graefe, G. (1990). Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. ACM SIGMOD Int'l. Conf.*, pages 749–764, Atlantic City, NJ, USA.

Graefe, G. (1993). Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170.

Hasan, W. and Motwani, R. (1994). Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelining Parallelism. In *Proc. Int'l. Conf. on Very Large Data Bases*, pages 36–47, Santiago, Chile.

Hasan, W. and Motwani, R. (1995). Coloring away communication in parallel query optimization. In *Proc. Int'l. Conf. on Very Large Data Bases*, pages 239–250, Zurich, Switzerland.

Hong, W. and Stonebraker, M. (1993). Optimization of Parallel Query Execution Plans in XPRS. *Distr. and Parallel Databases*, 1(1):9–32.

Manegold, S., Obermaier, J. K., and Waas, F. (1997). Load Balanced Query Evaluation in Shared-Everything Environments. In *Proc. European Conf. on Parallel Processing*, pages 1117–1124, Passau, Germany.

Manegold, S., Waas, F., and Kersten, M. L. (1998). On Optimal Pipeline Processing in Parallel Query Optimization. Technical Report INS-R9805, CWI, Amsterdam, The Netherlands.

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

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

Walton, C. B., Dale, A. G., and Jenevein, R. M. (1991). A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In *Proc. Int'l. Conf. on Very Large Data Bases*, pages 537–548, Barcelona, Spain.

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

Wilschut, A. N., Flokstra, J., and Apers, P. M. G. (1995). Parallel Evaluation of Multi-Join Queries. In *Proc. ACM SIGMOD Int'l. Conf.*, pages 115–126, San Jose, CA, USA.