Integrating I/O Processing and Transparent Parallelism — Toward Comprehensive Query Execution in Parallel Database Systems

Stefan Manegold Florian Waas

CWI · Kruislaan 413
1098 SJ Amsterdam · The Netherlands
⟨firstname.lastname⟩@cwi.nl

Abstract

Query processing in parallel database systems stands or falls by efficient resource usage including CPU scheduling, I/O processing and memory allocation. Up to now, most research has focussed on load balancing issues concerning several resources of the same kind only, i.e. balancing either CPU load or I/O load exclusively. In this paper, we present *floating probe*, a novel strategy to utilize parallel resources in a shared-everything environment efficiently. The key idea of floating probe is dynamic load balancing of CPU and I/O resources by overlapping I/O-bound build phase and CPU-bound probe phase of pipeline segments. The extent of interleaving is only limited by data dependencies. Simulation results show, that floating probe achieves considerably shorter execution times with less memory demands than conventional pipelining strategies.

Keywords parallel databases, parallel query processing, dynamic load balancing, efficient resource utilization

INTRODUCTION

Parallel query processing is the key to the performance improvements demanded by modern database applications. Pipelining parallelism is of particular interest since it is easier to control and less resource consuming than independent parallelism yet offering a huge potential of parallelism. Moreover, for linear query trees, pipelining is the only possibility to exploit inter-operator parallelism (Hasan and Motwani, 1994).

The two major aspects of pipeline processing that need to be considered carefully are the processor scheduling—the actual parallelization—and the I/O processing to support the scheduling.

So far, much work has been devoted to different processor scheduling strategies. Schneider and DeWitt study pipelining techniques on right-deep trees of hash join operators, proposing two distinct phases of processing (Schneider and DeWitt, 1990): Firstly, during the *build phase*, the inner relations of the joins are read from disk and hash tables are built in parallel. Secondly, during the *probe phase*, the outer relation is piped bottom-up through all operators.

Figure 1 shows an example for a *pipeline segment*. R_i and I_i denote the inner input relations and the intermediate results, respectively. I_1 denotes the outer input relation of the segment. Each input relation is either a base relation or the result of another pipeline segment. The R_i are all materialized on disk while I_1 is to be read from disk or received directly from another process.

Figure 2 depicts the functional decomposition of the example into build phase and probe phase. B_i denotes the operation to build the hash table H_i and P_i denotes the operation to probe I_i against H_i.

Considering also the physical memory limits, Chen et al. introduce a decomposition of the right-deep trees into pipeline segments, which fit into main memory (Chen et al., 1992). Thus, I/O caused by swapping can be avoided. The segments are evaluated one at a time with maximal computing resources at their disposal. The processor scheduling is a grouping of processors which are assigned to single operators according to work load estimations. Shekita et al. extend this method to capture also bushy operator trees by decomposing bushy trees into right-deep pipeline segments, thus, combining the flexibility of bushy operator trees with pipelining execution. Furthermore, not only join operators are considered but the more general notion of *blocking* and *non-blocking operators* is used: a pipeline segment is a sequence of non-blocking operators, which produce output on-the-fly—e.g. selection, projection without duplicate elimination, or the probe phase of either a hash join or a general index join. Only the last operator of the

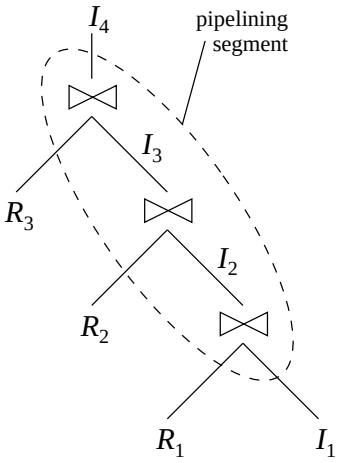

Figure 1: Pipelining segment

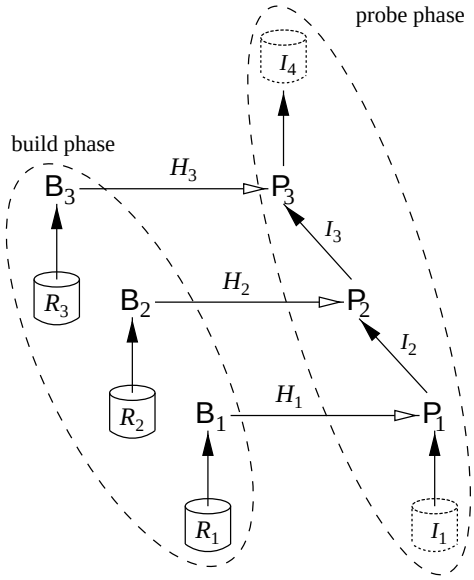

Figure 2: Build phase and probe phase

segment may be a blocking operator which has to collect all input before it produces any output—e.g. sort or aggregation operators.

Though offering the high potential for performance improvements, the aforementioned techniques proof a pitfall as soon as the actual query execution does not match the assumptions the preceding optimization was based on. Dynamic processor scheduling helps overcome this drawback. In (Manegold et al., 1997), we presented *DTE*, a strategy that serves as a transparent interface to pipelining parallelism by dynamically assigning processors to operators and thereby achieving a near-optimal exploitation of CPU resources. DTE avoids the major problems conventional pipelining suffers from: *discretization error* and *startup/shutdown delay* (Ganguly et al., 1992; Srivastava and Elsesser, 1993; Wilschut and Apers, 1991; Wilschut et al., 1995).

While DTE solves most of the critical issues in CPU scheduling, the I/O processing is ignored and hidden in the assumption that all hash tables are built before-hand, as supposed in previous work. In general, the build phase is I/O-bound—i.e. building a hash table takes less time than reading the base relation from disk—while the probe phase is CPU-bound as no intermediate results are to be materialized on disk. Consequently, the execution cannot reach its optimal performance due to inefficient resource

utilization: during the build phase the CPUs are idle, while during the probe phase the I/O system is idle.

To the authors' best knowledge—Hong is the only one addressing the integration of I/O processing. He proposes a scheduling algorithm that executes one CPU-bound and one I/O-bound task concurrently, to achieve a CPU-I/O-balanced workload in total (Hong, 1992). This in turn restricts the algorithm to scheduling of distinct data-independent tasks, only. Obviously, pipeline processing cannot benefit from this technique.

In this paper, we propose *floating probe*, a novel strategy to integrate both CPU scheduling and I/O processing on shared-everything platforms. We suppose that an optimizer has already generated a tree-shaped query plan and partitioned the plan in pipeline segments with the following characteristics: (1) Only the last operator of each segment might be a blocking operator, all other operators are non-blocking operators. The optimizer tuned the size of each segment that (2) all tables fit into main memory and (3) the probing then can be done without intermediate I/O (cf. (Chen et al., 1992; Schneider and DeWitt, 1990; Shekita et al., 1993)).

The key idea is to mesh build and probe phase as tightly as possibly, i.e. while conducting the probe for a group of operators, the hash tables of the successors can be loaded in the meantime. This procedure is subject to some constraints, e.g. during the very first build phase no probing can take place.

All segments are evaluated one after the other according to the producer/consumer data dependencies between them. We avoid parallel evaluation of data independent pipeline segments, as no performance improvements can be achieved that way (Shekita et al., 1993).

Floating probe establishes automatically balanced CPU and I/O workload throughout the whole execution, yielding not only shorter execution times in total but also lowering the memory requirements significantly.

Road-Map. The remainder of the paper is organized as follows. In the next section, we present the basic techniques for building the hash tables and discuss the two ways of parallelizing this step. Then, DTE, our strategy to implement the probe phase is introduced and discussed. We briefly point out the strength of DTE compared to conventional pipelining techniques in a representative selection of experiments. The next section concerns the problems occurring when both build and probe phase are combined. We present floating probe and show how I/O processing can be integrated with efficient CPU scheduling. The analysis of floating probe yields a near optimal upper bound. A simulation model and a comparative performance evaluation verifies the previously derived results. The work is concluded by a summary and a discussion of future work.

TABLE BUILDING PHASE

Shared-everything systems like SMPs provide uniform and parallel access to all attached disks. To exploit I/O parallelism we assume that each base relation is partitioned and fully declustered over all disks. Once, this is established, I/O parallelism utilizing the full I/O bandwidth can be used for every access to the base relations—even for exclusive access to a single relation. Furthermore, double buffering and asynchronous I/O allow an overlapping of CPU and I/O phases.

Building one single hash table in parallel

To build one single hash table in parallel,—i.e. using all disks and all CPUs—one thread per CPU, that reads tuples (one at a time) from a shared buffer pool, and inserts the tuple into the hash table, is started. Note, that CPU contention may occur if the number of threads exceeded the number of CPUs. Obviously, this strategy provides optimal load balancing.

The only problem occurring is to bridge the gap between the shared buffer pool and the disk I/O. As a simple solution to this problem, we extend one of the threads with some additional functionality: invoking asynchronous parallel I/O to read pages from disk. As the time to invoke the I/O of one page is by approximately three orders of magnitude smaller than the time to read a single page from disk, this additional task does not form a bottleneck.

In the reminder of this paper, we use Build(R_i) to denote the parallel building of the hash table that belongs to the i-th join within the pipeline. This includes reading R_i from disk using parallel I/O.

Building multiple hash tables in parallel

With these preliminaries, two different methods for building the hash tables become feasible: building all hash tables simultaneously and execute Build(R_1) through Build(R_N) concurrently, or executing only one single Build(R_i) at a time, i.e. executing Build(R_1) through Build(R_N) one after the other. Due to the full declustering of each base relation, both strategies can exploit the full I/O bandwidth. However, the first strategy would cause additional seek time as it has to cope with random disk access patterns when fragments of different relations—located on the same disk—are read concurrently. In contrast to this, the second strategy accesses larger homogeneous blocks and reduces the latency significantly. For this reason, we use the second strategy for our further considerations.

TUPLE PROBING PHASE

Our strategy to evaluate the probe phase of pipeline segments is *Data Threaded Execution (DTE)* (Manegold et al., 1997). In the reminder of this section, we first give a short overview of DTE and then we present a quantitative assessment of DTE.

The Model

The key idea of DTE is to dynamically assign the available processors to the data that is to be processed. We do this by gathering all operators of a pipeline segment into one stage and assigning all processors to this stage. This leads to optimal load balancing and efficient resource utilization without causing any additional overhead.

As it is not possible to perform two successive operators on the same input tuple in parallel, our approach is to switch from conventional operator parallelism to data parallelism. Data parallelism covers both, intra-operator (different tuples, same operator) and inter-operator (different tuples, different operators) parallelism.

To achieve this, DTE uses one thread per processor. Each thread is able to perform all operations within the active pipeline segment. The input tuples for the pipeline segment are provided in a global queue which can be accessed by all threads. Each thread takes one tuple at a time from the global input queue and guides it the way through all the operators of the pipeline segment 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 pipeline segment, 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, the thread takes 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 before it can proceed with the next input tuple from the queue. Figure 3 sketches the data threaded execution of a pipeline segment consisting of three joins on four processors.

There are no data dependencies between the threads. Thus, all threads start their processing simultaneously without any idle time, and none of them is idle until it finishes its work. In other words, there is no startup execution delay and there is no idle time due to synchronization among the processors. The only idle time that may occur is due to shutdown execution delay. As soon as a processor has finished its work and there are no more input tuples in the global queue, it is idle until the other processors have finished there work, too. This time is at most the time that one processor needs to process one tuple through the pipeline segment. In cases of extreme skew, the performance of DTE suffers from this shortcomming. We solved this problem by adding a simple but powerful redistribution mechanism to DTE yielding DTE/R (Manegold and Waas, 1998). As situations with extreme skew are not relevant in the context of this paper, we stick to the base version of DTE, for simplicity. The strategies presented in the remainder of this paper also apply to DTE/R.

DTE provides automatic and dynamic load balancing between the processors, as each thread can process the next input tuple as soon as it has finished the processing of the former tuple. Thus, all

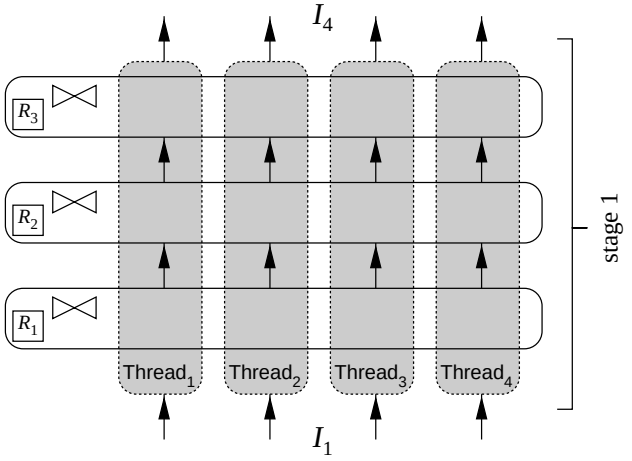

Figure 3: DTE

name	description	value
n	number of joins	1 to 16
$\|\|R_i\|\|$	cardinality of base relations	5k to 200k
v	range of join attribute values	$1 \leq v \leq \|\|R_i\|\|$
δ	attribute value distribution of join attributes	round-robin, uniform, normal1 (mean=$\frac{v}{2}$, dev.=$\frac{v}{10}$), normal2 (mean=$\frac{v}{2}$, dev.=$\frac{v}{5}$), exponential (mean=$\frac{v}{2}$)
af_i	augmentation factor of join i	$af_i = \frac{\|\|I_{i+1}\|\|}{\|\|I_i\|\|}$

Table 1: Query Parameters

processors are working as long as there are input tuples in the queue, i.e. neither startup delay nor discretization error occur with DTE. DTE outperforms conventional pipelining strategies significantly (Manegold et al., 1997).

In particular, this kind of load balancing—and thus efficient resource utilization—does not depend on cardinalities. Therefore, the efficiency of DTE does not suffer from any errors when estimating cardinalities and selectivities at compile time. If such errors lead to a non-optimal query tree, DTE cannot compensate this error but still provides a stable execution in the sense of efficient resource utilization without overhead, i.e. the situation cannot exacerbate any further.

Quantitative Assessment

In order to assess DTE quantitatively and to compare it to conventional pipeline execution (*PE*) as presented in the introduction on page 1 (for details see also (Manegold et al., 1997)), we implemented both strategies prototypically. Using this implementation, we ran several experiments on SGI PowerChallenge and Onyx shared-memory machines with 4 processors each.

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 take the median of 10 runs.

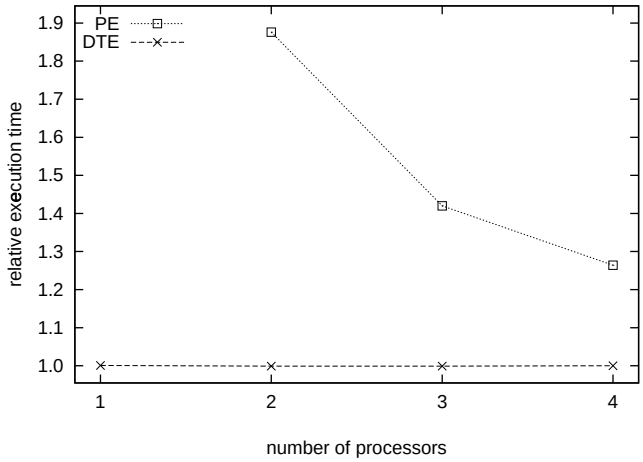

Figure 4: Wisconsin's joinAselB-query

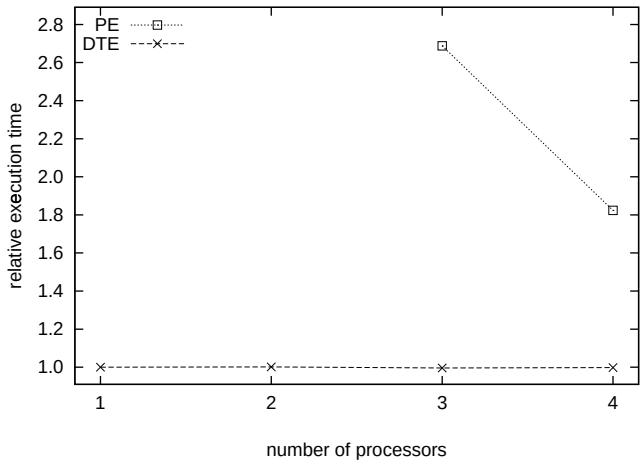

Figure 5: Wisconsin's joinCselAselB-query

Wisconsin Benchmark. The initial set of experiments deals with running two queries, namely joinAselB and joinCselAselB, of the Wisconsin Benchmark (Gray, 1993). We implemented the selection as semijoin, thus, joinAselB and joinCselAselB consist of pipeline segments of length 2 and 3, respectively.

Figures 4 and 5 depict the relative execution times $\frac{T_{\mathrm{PE}}}{T_{\mathrm{DTE}}}$ for joinAselB and joinCselAselB, respectively. PE performs substantially worse than DTE, mainly due to discretization error, as in both queries the first operator causes ten times as much work as the others. With DTE, each of the p processors used performs $\frac{1}{p}$-th of the total work. With PE, $p - x$ processors ($x = 1$ for joinAselB and $x = 2$ for joinCselAselB) do $\frac{10}{10+x}$-th of the total work, while x processors do $\frac{1}{10+x}$-th each.

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—again, 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. The execution time of PE is up to 2.2 times longer than

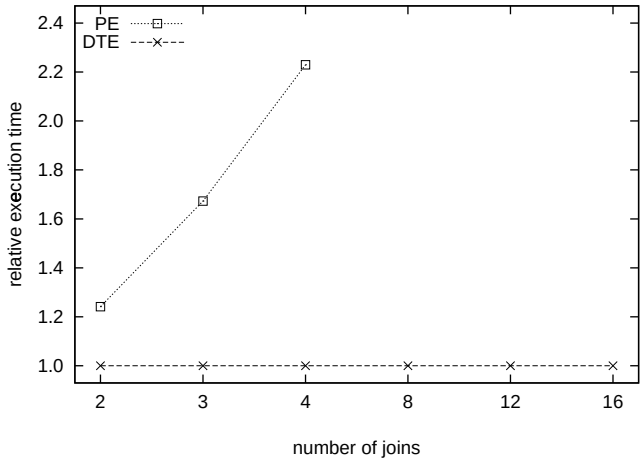

Figure 6: Average case (round-robin)

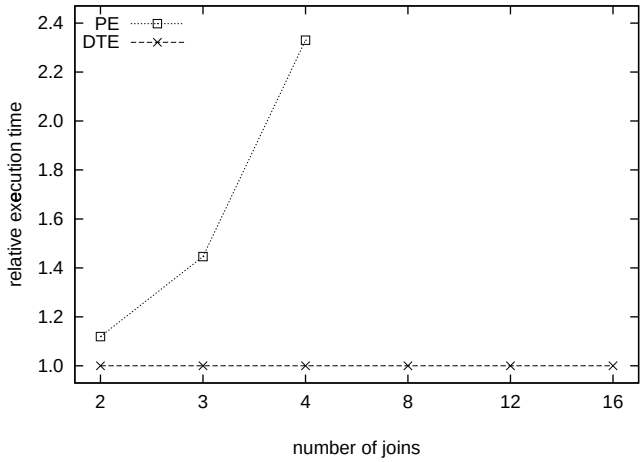

Figure 7: Average case (uniform)

that of DTE.

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

Speedup and Scaleup. Besides this overall performance comparison, we also ran experiments to measure the speedup and scaleup (DeWitt and Gray, 1992) of the different strategies. Figure 11 shows the speedup behavior of PE and DTE for a two-join-query with $af_1 = 1$ and $af_2 = 1/3$ (see Tab. 1). DTE provides near-linear speedup, whereas PE suffers from discretization error, obviously. Similarly, Figure 12 exemplary shows the scaleup behavior of PE and DTE for a two-join-query. We increased the weight of the pipeline segment with the number of processors by increasing af_2 appropriately while leaving $af_1 = 1$. DTE shows a negligible performance decrease of 1% when moving from one to two processors, but then, its 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.

The results obtained from the implementation of DTE are closed to our simulations results presented in (Manegold et al., 1997) showing the adequacy of our simulator.

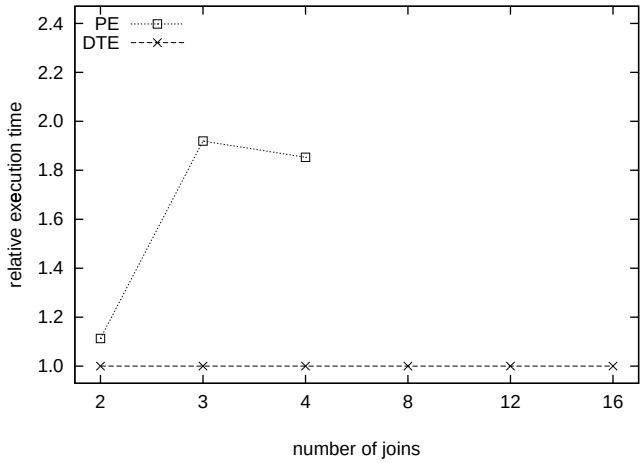

Figure 8: Average case (normal1)

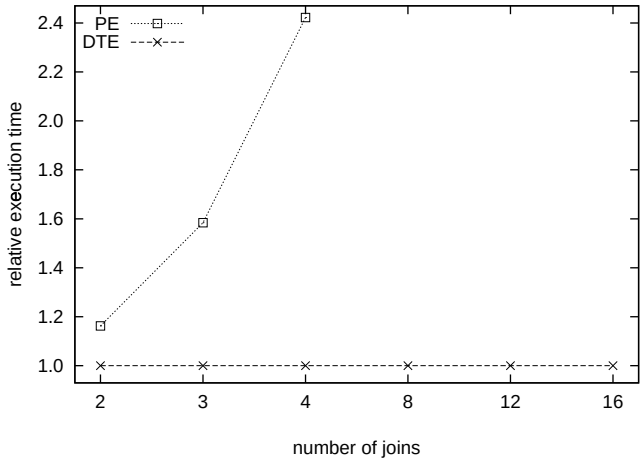

Figure 9: Average case (normal2)

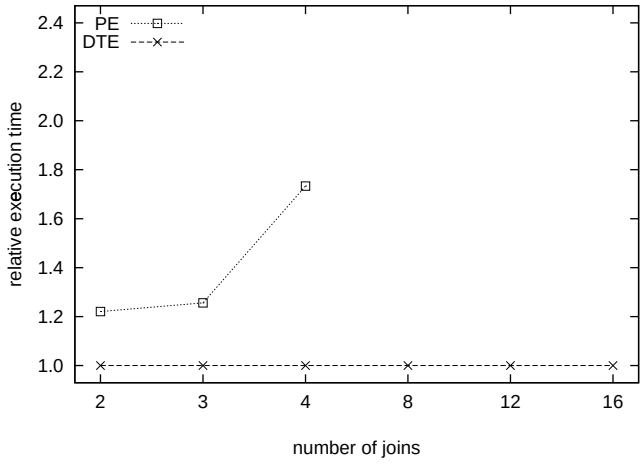

Figure 10: Average case (exponential)

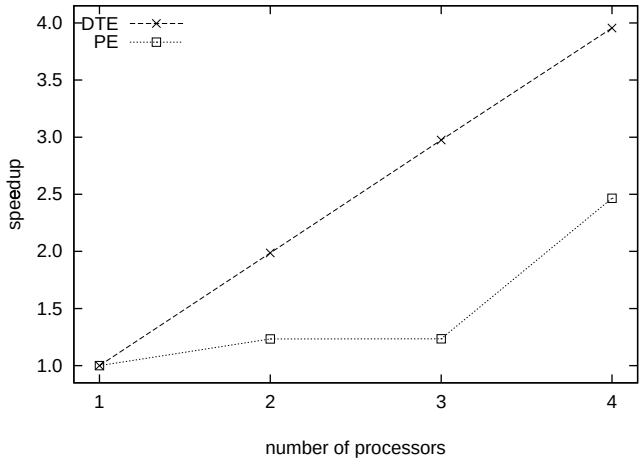

Figure 11: Speedup

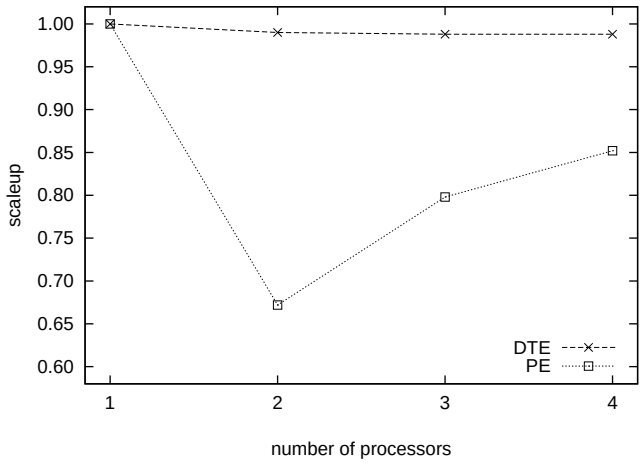

Figure 12: Scaleup

name	description	value
p	number of CPUs	1 - 8
d	number of disks	1 - 8
T_M	time to access one tuple in memory	10.0 μs
T_B	time per tuple to build a hash table	5.5 μs
T_P	time to probe one tuple against a hash table	4.0 μs
T_G	time to generate one result tuple	30.0 μs
T_I	time to invoke I/O for one block	7.4 μs
T_W	time to setup I/O system	1.0 ms
T_S	average I/O seek time	1.2 ms
bw	I/O bandwidth per disk	3 MB/s
bs	size of one I/O block in bytes	8 kB
T_R	$= \frac{bs}{bw}$, I/O time to read one block	2.6 ms
ts_R	size of tuples of relation R in bytes	100-200 Bytes
$\|\|R\|\|$	size of relation R in tuples	10^3 - $2 \cdot 10^5$
$\|R\|$	$= \left\lceil \frac{\|\|R\|\| \times ts_R}{bs} \right\rceil$, size of R in blocks	13 - 4883
N	number of joins	4 - 16

Table 2: Notation

BUILDING AND PROBING

Before we discuss the different strategies how to combine build phase and probe phase, we introduce further notation we use in the remainder of this paper. Build(R_i) (short B$_i$) denotes the building of the i-th hash table H_i (cf. page 3). This includes reading R_i from disk. Alloc(H_i) (A$_i$) denotes the allocation of memory for hash table H_i. Releasing the respective memory is denoted by Free(H_i) (F$_i$). Probe(I_i) (P$_i$) denotes the probing of intermediate result I_i through the i-th join within the pipeline using DTE. Probe($I_i..I_j$) (P$_{i..j}$) denotes the parallel probing of the joins i through j ($1 \leq i \leq j \leq N$) using DTE. Thus, both Probe(I_i) and Probe($I_{i..j}$) represent the execution of the respective subset of operators of the whole pipeline (Probe($I_1..I_N$)). Table 2 gives further notation and some basic cost values taken from literature. In Figure 13, we present the cost functions for single operations we use in the remainder of this paper.

Deferred Probe

A naive way to combine build and probe phase is to execute them one after the other: First, all hash tables are built, and after that, the probing is done (using DTE, in our case). We call this *deferred probe*.

The execution of the whole pipeline segment, i.e. build and probe phase, proceeds as follows: Alloc(H_1); Build(R_1); ... ; Alloc(H_N); Build(R_N); Probe($I_1..I_N$); Free(H_1); ... ; Free(H_N). For simplicity of presentation, we neglect the time consumed by Alloc(H_i) and Free(H_i). Assuming that CPU and I/O can overlap perfectly, the execution times of each single Build(R_i) as well as the execution time of Probe($I_1..I_N$) are given by the maximum of the corresponding CPU and I/O times. Thus, the total execution time of the whole pipeline segment is (cf. Fig. 13 and Tab. 2 for details):

$$
\begin{aligned}
T'_{\mathsf{defer}} &= T_{\mathsf{Build}} + T_{\mathsf{Probe}} \\
&= \sum_{i=1}^{N} \max\{O_s(R_i)\ ,\ C_B(R_i)\}\ + \\
&\quad \max\{O_r(I_1) + O_r(I_{N+1})\ ,\ C_{Px}(I_1..I_N)\}\,.
\end{aligned}
$$

I/O time without disk arm contention (sequential I/O):

$$O_s(R_i) \quad = \quad T_S + \left\lceil \frac{|R_i|}{d} \right\rceil (T_W + T_R)$$

I/O time with disk arm contention (random I/O):

$$O_r(R_i) \quad = \quad \left\lceil \frac{|R_i|}{d} \right\rceil (T_S + T_W + T_R)$$

CPU time to initialize I/O and to access a relation in memory:

$$C_x(I_i) \quad = \quad \left\lceil \frac{|I_i|}{p} \right\rceil T_I + \left\lceil \frac{||I_i||}{p} \right\rceil T_M$$

CPU time to build a hash table (incl. initialization of I/O):

$$C_B(R_i) \quad = \quad \left\lceil \frac{|R_i|}{p} \right\rceil T_I + \left\lceil \frac{||R_i||}{p} \right\rceil T_B$$

CPU time to probe a join:

$$C_P(I_i) \quad = \quad \left\lceil \frac{||I_i||}{p} \right\rceil T_P + \left\lceil \frac{||I_{i+1}||}{p} \right\rceil T_G$$

CPU time to probe joins (incl. fetching the input, storing the output and initialization of I/Os):

$$C_{Px}(I_i..I_j) \quad = \quad C_x(I_i) + C_P(I_i..I_j) + C_x(I_{j+1})$$

convenient abbreviation ($\Phi \in \{O_s, O_r, C_x, C_B, C_P\}$):

$$\Phi(R_i..R_j) \quad = \quad \sum_{k=i}^{j} \Phi(R_k)$$

Figure 13: Cost Functions

Suppose that either both phases are I/O-bound

$$\begin{aligned} & \forall i \in \{1, \ldots, N\} : && O_s(R_i) && > && C_B(R_i) \\ \wedge \quad & && O_r(I_1) + O_r(I_{N+1}) && > && C_{Px}(I_1..I_N) \end{aligned}$$

or both phases are CPU-bound

$$\begin{aligned} & \forall i \in \{1, \ldots, N\} : && O_s(R_i) && < && C_B(R_i) \\ \wedge \quad & && O_r(I_1) + O_r(I_{N+1}) && < && C_{Px}(I_1..I_N), \end{aligned}$$

then deferred probe provides minimal execution time:

$$\begin{aligned} T^{\min}_{\mathsf{defer}} \quad = \quad & \max\{ O_s(R_1..R_N) + O_r(I_1) + O_r(I_{N+1}) \ , \\ & \qquad C_B(R_1..R_N) + C_{Px}(I_1..I_N) \qquad \}. \end{aligned}$$

However, in most environments the build phase is I/O-bound while the probe phase is CPU-bound—at least if the pipeline is long enough—, i.e.

$$\begin{aligned} & \forall i \in \{1, \ldots, N\} : && O_s(R_i) && > && C_B(R_i) \\ \wedge \quad & && O_r(I_1) + O_r(I_{N+1}) && < && C_{Px}(I_1..I_N). \end{aligned} \qquad (\star)$$

In this case, deferred probe has one shortcoming: Resources are not used as efficiently as theoretically possible. During the build phase, CPU capacities are left idle, while during the probe phase, I/O capacities are not fully used. Thus, the execution time is not optimal:

$$T_{\mathsf{defer}} \quad = \quad O_s(R_1..R_N) + C_{Px}(I_1..I_N) \quad \overset{(\star)}{>} \quad T^{\min}_{\mathsf{defer}}.$$

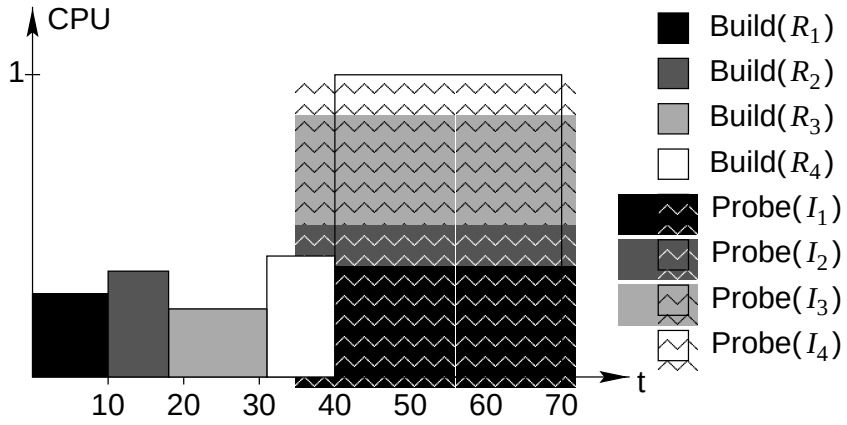

Figure 14: Sample CPU load (deferred probe)

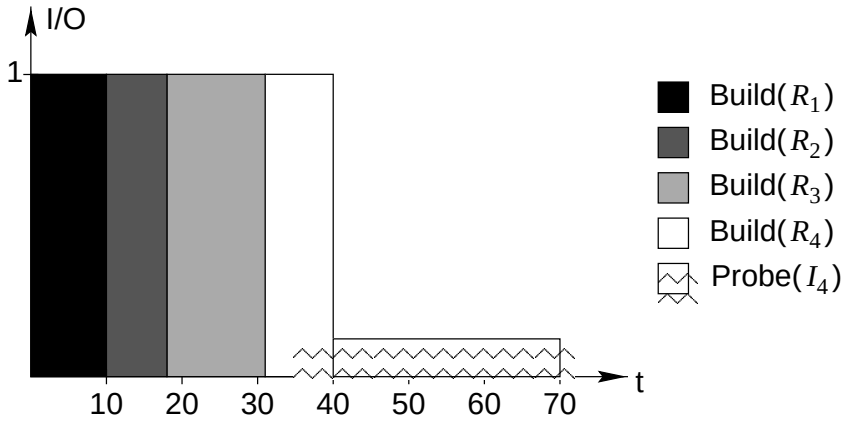

Figure 15: Sample I/O load (deferred probe)

Figures 14 and 15 depict CPU and I/O load of deferred probe evaluating a pipeline segment with four joins.

Multi-user and multi-query environments may balance the utilization of CPU and I/O. But these environments suffer form the exhaustive use of memory of deferred probe. The memory for the hash tables is allocated—possibly long time—before the hash tables are used in the probe phase and all memory is released only after the whole pipeline is executed (cf. Fig. 16).

In multi-user or multi-query environments, not only execution time (T) and maximal memory usage (m) should be regarded, but also the *memory usage area* (M = amount of memory usage × time memory is occupied).

Floating Probe

To overcome the shortcomings of deferred probe, our approach is to let the build phase and the probe phase overlap. Opposed to deferred probe, this results in a single phase that integrates build and probe phase. Thus, resource utilization can be balanced by combining I/O-bound build and CPU-bound probe. We call our new strategy *floating probe*.

The point is, $\mathsf{Probe}(I_i)$ can be started as soon as $\mathsf{Build}(R_i)$ has finished, i.e. $\mathsf{Probe}(I_i)$ can be executed in parallel with $\mathsf{Build}(R_{i+1})$. Thus, compared to deferred probe, some of the probe work is done before the build of the last hash table is finished. As building the hash tables is I/O-bound the elapsed time until all hash tables are build cannot be reduced. However, the probe work that has to be done after the last build is reduced and so is the overall execution time.

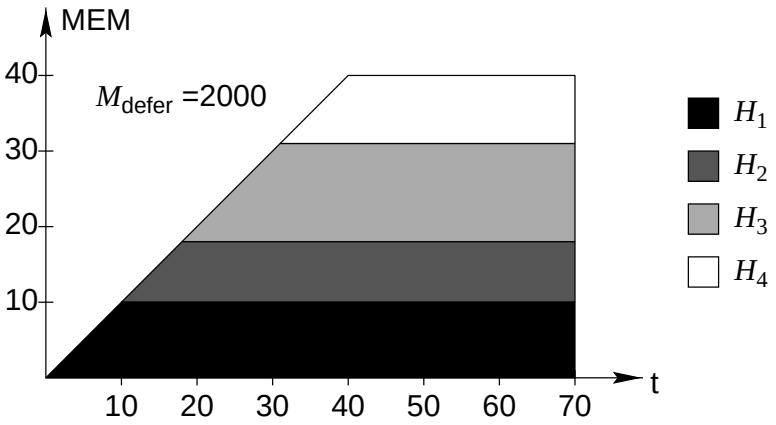

Figure 16: Sample memory usage (deferred probe)

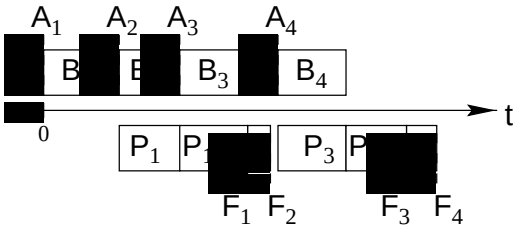

Figure 17: Sample schedule floating probe

Two cases have to be distinguished first: Either Probe(I_1) is CPU-bound—e.g. I_1 already resides in memory, is received via a fast network, or even reading from disk is faster than performing the probing—or Probe(I_1) is I/O-bound, i.e. reading I_1 from disk is slower than performing the probing.

Probe(I_1) is CPU-bound. In this case, floating probe proceeds as follows (cf. Fig. 17 for a sample schedule): At the beginning, the hash table H_1 of the first join is built (Build(R_1)). Thereafter, Probe(I_1) and Build(R_2) are started simultaneously and executed concurrently. As the output tuples produced by Probe(I_1) cannot yet be processed by Probe(I_2), they have to be buffered. To avoid intermediate I/O, this is done in memory. If Probe(I_1) ends before Build(R_2), H_1 is dropped. Otherwise, as soon as Build(R_2) has finished, Build(R_3) is started and the probe is extended, so that the remaining tuples of I_1 are piped through both probes (Probe($I_1..I_2$)). As before, the output of Probe($I_1..I_2$) is buffered in memory. If then Probe($I_1..I_2$) ends before Build(R_3), H_1 is dropped and the part of I_2 buffered in memory during Build(R_2) is processed through Probe(I_2). Otherwise, the probe is extended to Probe($I_1..I_3$), as soon as Build(R_3) is done. This proceeds until the last hash table H_N is built. After that, only probing is done until all tuples are processed: For each I_i that is partly buffered in memory Probe($I_i..I_N$) is executed. Figure presents floating probe as pseudo code. The Procedures that are used here and with the pseudo codes of the following strategies are presented in Figure 18.

With floating probe, the pipeline segment is dynamically extended to the next join once its hash table is built. Thus, allocated memory is used as soon as possible. On the other hand, hash tables are dropped immediately after the respective probe is done. Thus, allocated memory is released as soon as it is no longer needed.

Figures 20 and 21 depict CPU and I/O load of floating probe evaluating a pipeline segment with four joins—I_1 is receive via the network and I_5 is written to disk—and Figure 22 shows the respective memory usage.

Probe(I_1) is I/O-bound. Now, consider the case reading I_1 from disk is slower than performing the probe. As I_1 is also fully declustered across all disks, there is no sense in running Probe(I_1) and Build(R_2)

```
procedure Init() do  // initialization of global variables
    toBuild[1..N]   := {1, .., 1};  // part of H_i that has to be built
    toProbe[1..N]   := {1, .., 1};  // part of I_i that has to be probed
    allocated[1..N] := no;          // memory for H_i allocated ?
    next            := 1;           // next H_i that has to be built
    first           := 1;           // first I_i that has to be probed
    last            := 0;           // last I_i that can be probed
    built           := 0;           // part of H_i that has been built
    probed          := 0;           // part of I_i that has been probed
od;

procedure BuildOnly(R_next) do
    if allocated[next] = no then Alloc(H_next); allocated[next] := yes; fi;
    Build(R_next); toBuild[next] := 0; next ++; last ++;
od;

procedure ProbeOnly(I_first..I_last) do
    Probe(I_first..I_last); probed := toProbe[first];
    foreach i ∈ {first, ..., last} do toProbe[i] -= probed; od;
    while toProbe[first] = 0 ∧ first ≤ N do first ++; Free(H_first); od;
od;

procedure BuildAndProbe(R_next, I_first..I_last) do
    if allocated[next] = no then Alloc(H_next); allocated[next] := yes; fi;
    do built := Build(R_next) || probed := Probe(I_first..I_last);
    until first of both ends;
    foreach i ∈ {first, ..., last} do toProbe[i] -= probed; od;
    while toProbe[first] = 0 ∧ first ≤ N do first ++; Free(H_first); od;
    toBuild[next] -= built;
    if toBuild[next] = 0 then next ++; last ++; fi;
od;
```

Figure 18: Procedures

in parallel due to disk access contention. We present two strategies, how to proceed in this case.

The first is to defer Probe(I_1) until enough, say g, hash tables are built, such that executing Build(R_{g+1}) and Probe($I_1..I_g$) concurrently is approximately CPU-I/O-balanced, or at least such that executing Probe($I_1..I_g$) is CPU-bound. Thus, running Probe(I_1) I/O-bound is avoided. But on the other hand, the start of probing is deferred and Build(R_2) through Build(R_g) are run I/O-bound. As soon as Probe($I_1..I_g$) is started, execution continues as usual. We call this strategy *late probing* (cf. Fig. 23a).

The second strategy is to execute Probe(I_1) right after Build(R_1), materializing I_2 completely in memory, and to defer Build(R_2) until Probe(I_1) is done. As soon as Build(R_2) has finished, processing resumes with starting Build(R_3) and Probe(I_2) simultaneously. Thus, Probe(I_1) is run I/O-bound as well as Build(R_2) thereafter. But on the other hand, probing is started as soon as possible. We call this strategy *early probing* (cf. Fig. 23b).

The case, that the result relation of the pipeline segment is not kept in memory, but rather written to disk, does not need any special treatment. Probe(I_N) can only be processed after Build(R_N) is done. Hence, there is no I/O interference.

```
begin
   Init();
   do
      if next ≤ N then
         if first ≤ last then
            BuildAndProbe(R_next, I_first..I_last);
         else /* first > last */
            BuildOnly(R_next);
         fi;
      else /* next > N */
         ProbeOnly(I_first..I_last);
      fi;
   until first > N;
end.
```

Figure 19: Floating probe (CPU-bound Probe(I_1))

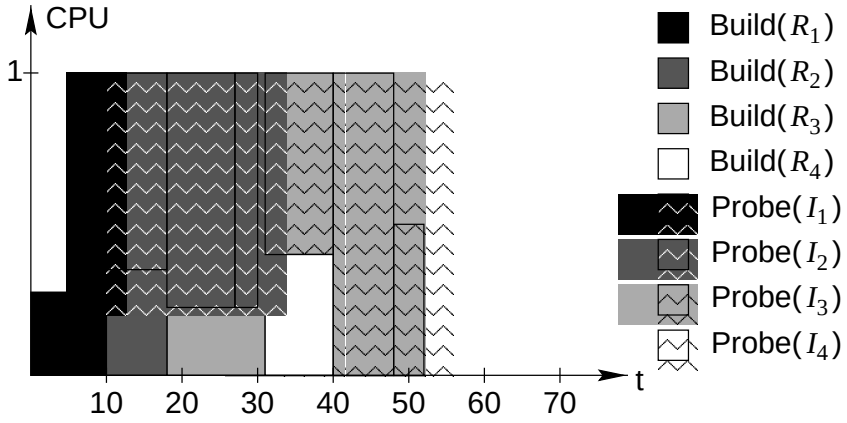

Figure 20: Sample CPU load (floating probe)

The first advantage of floating probe is that the overall execution time is reduced as some of the probe work is done before Build(R_N) has finished. In our example, deferred probe needs 70 units of time,

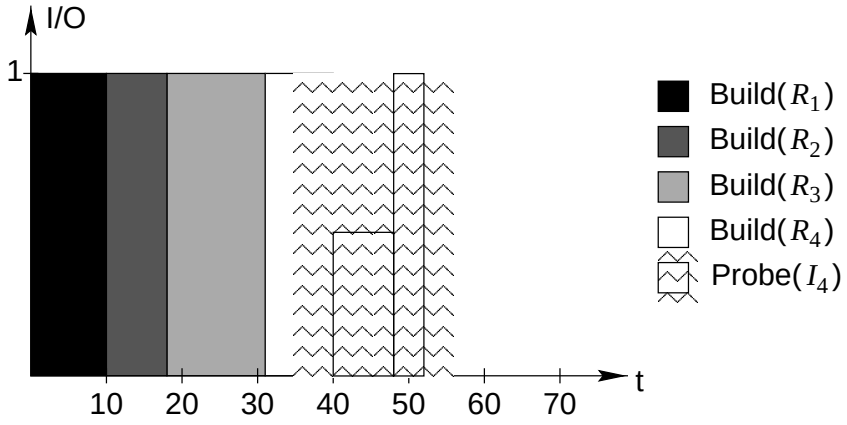

Figure 21: Sample I/O load (floating probe)

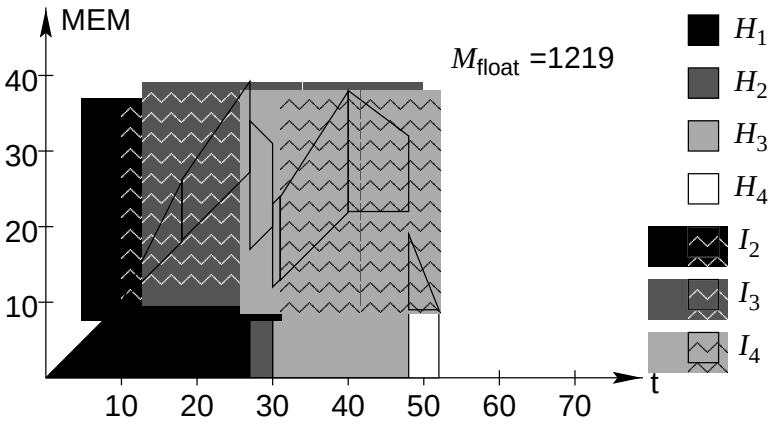

Figure 22: Sample memory usage (floating probe)

```
Init();
do
    BuildOnly(R_next);
until ProbeOnly(I_first..I_last) is I/O-bound;
```

a) Replacement for Init() in Fig. : *late probing*

```
Init();
BuildOnly(R_next);          // next = 1
ProbeOnly(I_first..I_last); // first = last = 1
```

b) Replacement for Init() in Fig. : *early probing*

Figure 23: Floating probe (I/O-bound Probe(I_1))

whereas floating probe needs only 52 (cf. Figs. 14,15,20,21). There is a lower bound, as the execution time cannot be less than needed to do the total work without any overhead or synchronization. This bound is

$$\begin{aligned} T^{\min}_{\mathsf{float}} &= \max\{O_s(R_1..R_N) + O_s(I_1) + O_s(I_{N+1}) \ , \\ &\qquad C_B(R_1..R_N) + C_{Px}(I_1..I_N) \quad \} \\ &\overset{(\star)}{<} O_s(R_1..R_N) + C_{Px}(I_1..I_N) \ = \ T_{\mathsf{defer}}. \end{aligned}$$

Obviously, if either building or probing dominates the overall execution costs, i.e. either $O_s(R_1..R_N) \gg C_{Px}(I_1..I_N)$ or $C_{Px}(I_1..I_N) \gg O_s(R_1..R_N)$, then floating probe cannot perform much better than deferred probe. Further, the minimal execution time of floating probe cannot be less than half the execution time of deferred probe:

$$\begin{aligned} T^{\min}_{\mathsf{float}} &\overset{\dagger}{\geq} \max\{O_s(R_1..R_N) \ , \ C_{Px}(I_1..I_N)\} \\ &\overset{\ddagger}{\geq} \quad \frac{O_s(R_1..R_N) + C_{Px}(I_1..I_N)}{2} \quad = \ \frac{T_{\mathsf{defer}}}{2}. \end{aligned} \qquad (\star\star)$$

Here, equality holds, (†) iff $O_s(I_1) = O_s(I_{N+1}) = 0 \ \wedge \ O_s(R_1..R_N) \geq C_B(R_1..R_N) + C_{Px}(I_1..I_N)$, and (‡) iff $O_s(R_1..R_N) = C_{Px}(I_1..I_N)$.

The second advantage of floating probe is reduced memory consumption. If any probe finishes before Build(R_N) is done, the corresponding hash table is released, and thus, the memory usage area M (cf.

page 12) is smaller than that of deferred probe. In our example, the memory usage area of deferred probe amounts to 2000 units, whereas floating probe needs only 1219 units (cf. Figs. 16 & 22).

A drawback of floating probe is, that parts of intermediate results have to be materialized in memory. This causes additional CPU costs and additional memory is needed. But the results of our simulation experiments show, that floating probe outperforms deferred probe, despite these overheads.

Neglecting these overheads—and most of the synchronization that arises due to data dependencies—for the moment, the execution time of floating probe is:

$$
\begin{aligned}
T_{\mathsf{float}} &= \max\{O_s(R_1)\ ,\ C_B(R_1)\} + \\
&\quad \max\{O_s(R_2..R_N) + O_s(I_1)\ , \\
&\qquad C_B(R_2..R_N) + C_x(I_1) + C_P(I_1..I_{N-1})\} + \\
&\quad \max\{O_s(I_{N+1})\ ,\ C_P(I_N) + C_x(I_{N+1})\} \\
&\overset{(\star)}{=} O_s(R_1) + \\
&\quad \max\{O_s(R_2..R_N) + O_s(I_1)\ , \\
&\qquad C_B(R_2..R_N) + C_x(I_1) + C_P(I_1..I_{N-1})\} + \\
&\quad \max\{O_s(I_{N+1})\ ,\ C_P(I_N) + C_x(I_{N+1})\}
\end{aligned}
$$

ANALYSIS

According to the presentation of floating probe in the previous section, it seems to be rather complicated to implement this strategy, as a lot of explicit scheduling overhead is necessary. In the following, we discuss a rather simple but effective method to avoid this scheduling overhead and describe our simulation model. Thereafter, we present the results of our experiments comparing deferred probe and floating probe.

Simulation Model

Although both phases are no longer executed one after the other, they are still in some sense independent of each other. The only dependency between the two phases is that a hash table has to be built before the respective intermediate result can be probed against it. Thus, our solution is to implement the build phase and the probe phase with separate threads. The only communication between build thread and probe thread is that the build thread has to inform the probe thread as soon as it has built a hash table. Using this information, the probe thread can decide, whether it can probe the current tuple through the next join or whether it has to materialize it as the next hash table is not yet built. Both threads are started concurrently. To guarantee, that the probe thread only uses those CPU resources that are not used by the build thread, the probe thread is run with lower priority than the build thread. Using this implementation technique, scheduling is done by the operation system.

In order to compare floating probe to deferred probe, we designed and implemented an event driven simulator using the Sim++ package (Fishwick, 1995). The simulator is very detailed, i.e. it simulates each single page-I/O-operation as well as each single tuple-operation using the execution times from Table 2. According to the aforementioned strategy, the simulator assumes distinct build and probe threads, one of each per processor.

Experiments

We randomly generated pipeline segments of several classes. Each class is characterized by the length $N \in \{4, 8, 16\}$ of the pipeline segment and the location L of I_1 and I_{N+1}. $L(I_1) = \texttt{disk}$ means that I_1 is initially stored on disk and $L(I_1) = \texttt{net}$ means that I_1 is received via network. Analogously, $L(I_{N+1}) = \texttt{disk}$ means that I_{N+1} finally has to be stored on disk and $L(I_{N+1}) = \texttt{net}$ means that I_{N+1} is sent to the network. The location of I_{N+1} affects both strategies equally: If $L(I_{N+1}) = \texttt{disk}$, in both strategies I/O is needed during $\mathsf{Probe}(I_N)$, i.e. after $\mathsf{Build}(I_N)$ is done. If $L(I_{N+1}) = \texttt{net}$, however, no I/O is needed during $\mathsf{Probe}(I_N)$ in either strategy. For this reason, we restrict our discussion here to

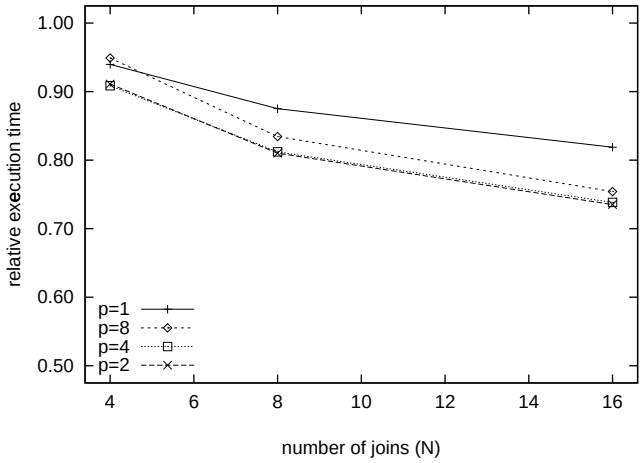

Figure 24: $\overline{T}_{\mathtt{f/d}}(\mathtt{disk}, N, p)$

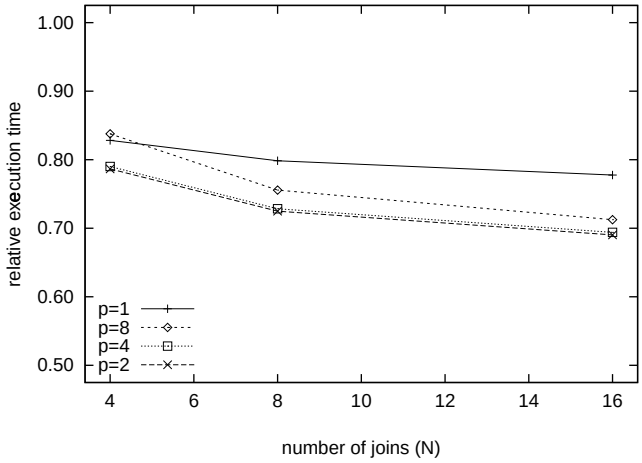

Figure 25: $\overline{T}_{\mathtt{f/d}}(\mathtt{net}, N, p)$

the two cases that either $L(I_1) = L(I_{N+1}) = \mathtt{disk}$, or $L(I_1) = L(I_{N+1}) = \mathtt{net}$. In the second case, no I/O is needed to evaluate the probe phase. The results for the remaining two cases are similar to those presented.

We randomly generated 360 different segments for each class, with tuple sizes between 100 and 200 bytes and relation sizes between 10^3 and $2 \cdot 10^5$ tuples. All segments fulfilled condition ($\star$) on page 11.

For each segment $S_j^{L,N}$, we simulated the execution with both deferred probe and floating probe for different degrees of parallelism ($p \in \{1, 2, 4, 8\}$, $d = p$). If I_1 and I_{N+1} were located on disk, we simulated the execution for both variants of floating probe, early probing and late probing. The differences between both variants were not significant, thus, we present only those for late probing here. To compare the performance of deferred probe and floating probe, we calculated the relative execution time $T_{\mathsf{float}}(S_j^{L,N}, p)/T_{\mathsf{defer}}(S_j^{L,N}, p)$. Within each class—identified by L, N, and p—we calculated the average relative execution time over all the $n = 360$ queries:

$$\overline{T}_{\mathtt{f/d}}(L, N, p) = \frac{1}{n} \sum_{j=1}^{n} \frac{T_{\mathsf{float}}(S_j^{L,N}, p)}{T_{\mathsf{defer}}(S_j^{L,N}, p)}.$$

Figures 24 and 25 show the average relative execution times with ($L = \mathtt{disk}$) and without probe-I/O ($L = \mathtt{net}$), respectively. Floating probe outperforms deferred probe in any case ($\overline{T}_{\mathtt{f/d}}(L, N, p) < 1$). The

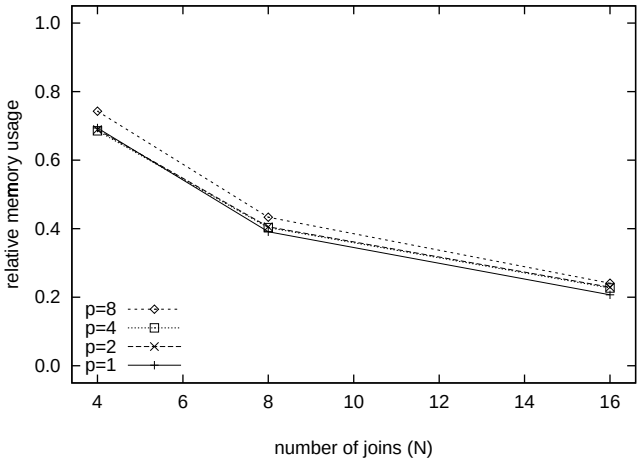

Figure 26: $\overline{M}_{\mathtt{f/d}}(\mathtt{disk}, N, p)$

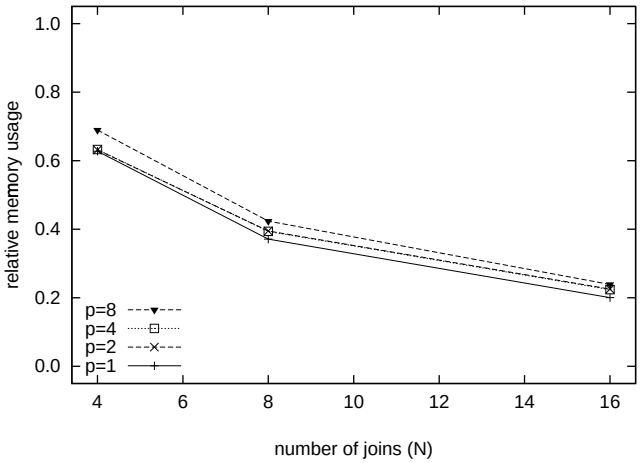

Figure 27: $\overline{M}_{\mathtt{f/d}}(\mathtt{net}, N, p)$

improvement increases with the length of the pipeline segment, as then the contribution of $\mathsf{Build}(R_1)$ and $\mathsf{Probe}(I_N)$—no improvement is possible during these operations due to data dependencies—to the total execution time becomes relatively small. Further, the results show that the performance gain of floating probe over deferred probe is bigger if no probe-I/O is needed. This is obvious, as without probe-I/O, more probe work can be done concurrently with the build.

Using floating probe instead of deferred probe saves up to 27% for $L = \mathtt{disk}$ and up to 31% of execution time for $L = \mathtt{net}$. Remember, that at most 50% can be saved (cf. ($\star\star$) on page 16). The average saving amounts to approximately 16% for $L = \mathtt{disk}$ and 24% for $L = \mathtt{net}$.

In addition to the execution times, we also examined the memory usage of floating probe and deferred probe. During the simulation, we calculated the total memory usage $M(S_j^{L,N}, p)$. Analogous to the average relative execution time, we calculated the average relative memory usage $\overline{M}_{\mathtt{f/d}}(L, N, p)$. Figures 26 and 27 show the results with ($L = \mathtt{disk}$) and without probe-I/O ($L = \mathtt{net}$), respectively. Again, floating probe performs better—i.e. needs less memory—than deferred probe. Here, the differences between $L = \mathtt{disk}$ and $L = \mathtt{net}$ are negligible. Floating probe saves up to 80% (55% on average) of memory allocation compared to deferred probe.

CONCLUSION

In this paper we addressed the topic of efficient resource utilization to boost query execution in parallel database systems. We presented *floating probe*, a new technique to evaluate pipeline segments in shared-everything environments which overcomes severe drawbacks of former methods. Floating probe balances the CPU and I/O workload between the I/O-bound build phase and the CPU-bound probe phase of pipeline segments optimally with respect to the data dependencies between both phases. Furthermore, floating probe (1) provides shorter execution times and (2) consumes less memory than deferred probe. Floating probe achieves these improvements without explicit scheduling, thus, floating probe neither needs any a priori cost estimations nor does it cause any scheduling overhead.

These properties make floating probe an easy-to-control and transparent means of parallelism: only the size of the pipeline segment and the degree of parallelism have to be determined by the optimizer while the execution strategy then guarantees the best execution possible. Floating probe is a building block to comprehensive query execution in parallel databases.

In the future, we will focus on the investigation of other means to incorporate further optimization decisions into the execution technique and, therefore, liberating the optimizer from tasks like finding the appropriate length of a pipeline segment.

Acknowledgments

We thank Johann K. Obermaier for his comments on a draft version of this paper.

References

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

Fishwick, P. A. (1995). *Simulation Model Design and Execution.* Prentice Hall, Englewood Cliffs, NJ, USA.

Ganguly, S., Hasan, W., and Krishnamurthy, R. (1992). Query Optimization for Parallel Execution. In *Proc. ACM SIGMOD Int'l. Conf.*, pages 9–18, San Diego, CA, USA.

Gray, J., editor (1993). *The Benchmark Handbook.* Morgan Kaufmann, San Mateo, CA, USA.

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.

Hong, W. (1992). Exploiting Inter-Operation Parallelism in XPRS. In *Proc. ACM SIGMOD Int'l. Conf.*, pages 19–28, San Diego, CA, USA.

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. and Waas, F. (1998). Thinking Big in a Small World — Efficient Query Execution on Small-scale SMPs. In *High Performance Computing Systems and Applications.* Kluwer Academic Publishers, Edmonton, AL, Canada.

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.

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

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.

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.