EFFICIENT RESOURCE UTILIZATION IN SHARED-EVERYTHING ENVIRONMENTS[†‡]

Stefan Manegold
CWI
P.O.Box 94079
1090 GB Amsterdam
The Netherlands
Stefan.Manegold@cwi.nl

Johann K. Obermaier
ABB Corporate Research Ltd
Computer Engineering Dept.
Information Technology CHCRC.C2
5405 Baden-Dättwil
Switzerland
Johann.Obermaier@chcrc.abb.ch

ABSTRACT

Efficient resource usage is a key to achieve better performance in parallel database systems. Up to now, most research has focussed on balancing the load on several resources of the same type, i.e. balancing either CPU load or I/O load. In this paper, we present *floating probe*, a strategy for parallel evaluation of pipelining segments in a shared-everything environment that provides dynamic load balancing between CPU- and I/O-resources. The key idea of floating probe is to overlap—as much as possible with respect to data dependencies—I/O-bound build phase and CPU-bound probe phase of pipelining segments to improve resource utilization. Simulation results show, that floating probe achieves shorter execution times while consuming less memory than conventional pipelining strategies.

1 INTRODUCTION

Parallel processing in database systems is a key to the required performance improvements of modern database applications.

Pipelining parallelism is of particular interest as it is much easier to control than *independent parallelism* and as no intermediate results need to be materialized. Additionally, for linear query trees, only pipelining is feasible to exploit inter-operator parallelism (Hasan and Motwani, 1994). Schneider and DeWitt (1990) study the effect of pipelining on a right-deep tree of hash join operators in detail. The evaluation of queries is split into two distinct phases. First, the inner relations are read from disk, and hash tables are built in parallel (*build phase*). Second, the outer relation is piped bottom-up through all operators (*probe phase*).

To avoid I/O, the right-deep tree is decomposed into segments, which fit in main memory (Chen et al., 1992). Segments are evaluated one at a time with maximal computing resources. Processors are assigned to the operators of a segment based on work estimations. This approach achieves pipelining parallelism between intra-parallel operators.

Shekita et al. expand this idea to bushy operator trees (Shekita et al., 1993). The bushy tree is disjointed in right-deep *pipelining segments*. Each pipelining segment consists of a sequence of *non-blocking operators*, which produce output on-the-fly, like selection, projection (without duplicate elimination), or the probe phase of either a hash join (for equi-joins) or a general index join (for θ-joins). Only the last operator in the sequence might be a *blocking operator* which has to collect all input before it produces any output, e.g. sort or aggregation. For each segment pipelining parallelism can be exploited. This combines the flexibility of bushy operator tree with pipelining execution.

Finally, in (Manegold et al., 1997), we presented DTE, a new strategy to execute the probe phase of pipelining segments in shared-everything environments. DTE avoids the major problems that 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). Further, DTE is resistant against execution skew and provides optimal execution by switching from operator parallelism to data parallelism.

But still one problem with the execution of pipelining segments remains open: In typical database environments, the build phase is I/O-bound (i.e. building a

[†]This work was partly supported by the German Research Council under contract DFG Fr 1142/1-1.

[‡]An extended version of this paper is available as (Manegold and Obermaier, 1997).

hash table takes less time than reading the base relation from disk) while the probe phase is CPU-bound (as no intermediate results are materialized on disk due to pipelining). Thus, execution cannot be optimal due to inefficient resource utilization: During the build phase the CPUs are idle, while during the probe phase the I/O system is idle.

Hong presents a scheduling algorithm that executes one CPU-bound and one I/O-bound task concurrently, to achieve a CPU-I/O-balanced workload (Hong, 1992). This algorithm is restricted to scheduling distinct data-independent task (i.e. distinct operators or pipelining segments), whereas we focus on executing the two data-dependent phases of one segment.

The contribution of this paper is *floating probe*, a new strategy to combine I/O-bound build phase and CPU-bound probe phase. Floating probe improves resource utilization by letting both phases overlap as much as possible, and thus automatically balancing CPU- and I/O-workload during evaluation. The benefits of our new strategy are twofold: First, floating probe provides shorter execution times than executing build and probe phase one after another. Additionally, floating probe requires less memory during execution than the traditional strategy.

The remainder of the paper is organized as follows. In Section 2, we define the problem we focus on. Our strategy to evaluate the build phase is described in Section 3. In Section 4, we present DTE, a strategy for efficient evaluation of the probe phase. Section 5 studies the problems that occur when combining both phases and presents our solution floating probe. A simulation model and a comparative performance evaluation is given in Section 6. Section 7 contains our conclusion.

2 THE PROBLEM

In this paper, we focus on the issue of load balanced execution of pipelining segments in shared-everything environments. We suppose that an optimizer has already generated a tree-shaped query plan and partitioned the plan in pipelining 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 necessary tables fit into in 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)).

Figure 1a depicts a sample pipelining segment consisting of three joins. 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

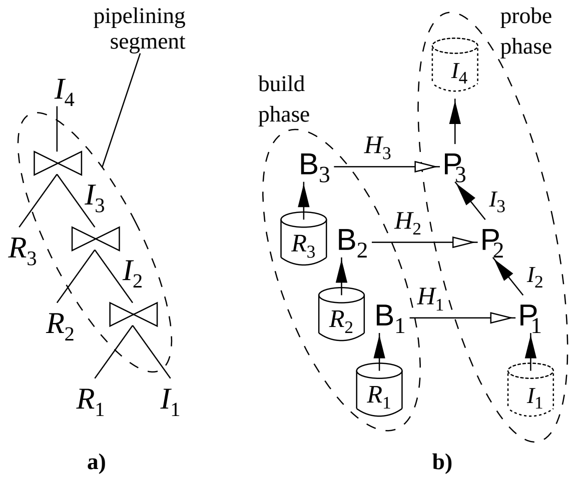

Fig. 1 Pipelining segment; build & probe phase

input relation is either a base relation or the result of an other pipelining segment. All R_i are materialized on disk. I_1 is either materialized on disk, or received on the fly from the network.

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 pipelining segments, as this obtains no performance improvements (Shekita et al., 1993). Evaluation of a segment proceeds in two phases: The first phase loads all inner relations of the segment and builds the (hash) indices (*build phase*). The second phase pipes all tuples of the outer relation through the probe phases of the joins (*probe phase*).

Figure 1b depicts the build phase and the probe phase of the sample segment. B_i denotes the operation to build the hash table H_i, while P_i denotes the operation to probe I_i against H_i.

3 TABLE BUILDING PHASE

Shared-everything systems provide uniform and parallel access to all disks. We assume that each base relation is partitioned and full declustered across all disks. Thus, full I/O parallelism—i.e. full I/O bandwidth—can be used even when accessing only a single relation. Further, double buffering and asynchronous I/O are used, so that CPU and I/O can overlap.

In the reminder of this paper, we use $\mathsf{Build}(R_i)$ 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.

There are two strategies to build all the hash tables of a pipelining segment. The first is to start building all hash tables simultaneously and execute $\mathsf{Build}(R_1)$

through Build(R_N) concurrently. The second strategy is to execute only one single Build(R_i) at a time, i.e. to execute Build(R_1) through Build(R_N) one after the other. Due to full declustering of each base relation, both strategies can exploit the full I/O bandwidth. But the first strategy would lead to additional seek time due to random I/O, as partitions of different relations (located on the same disk) are accessed concurrently. The second strategy outperforms the first one under these assumptions. Thus, we prefer the second strategy.

4 TUPLE PROBING PHASE

Our strategy to evaluate the probe phase of pipelining segments is *Data Threaded Execution (DTE)* (Manegold et al., 1997). In the reminder of this section, we give an short overview of DTE.

DTE uses one thread per processor. Each thread is able to perform all operations within the active pipelining segment. The input tuples for the pipelining segment are provided in a global queue that all threads can access. Each thread takes one tuple at a time from the global input queue and guides it the way through all the operators of the pipelining 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 pipelining 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 first, before it can proceed with the next input tuple from the queue.

DTE provides automatic and dynamic load balancing, and thus achieves efficient resource utilization. DTE outperforms conventional pipelining strategies significantly (Manegold et al., 1997).

5 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. Alloc(H_i) (short A$_i$) denotes the allocation of memory for H_i. Releasing the respective memory is denoted by Free(H_i) (F$_i$). Probe(I_i) (P$_i$) denotes the probing of 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 ($1 \le i \le j \le N$) using DTE.

Table 1 Notation

name	description	value
N	number of joins	
p	number of CPUs	
d	number of disks	
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	
ts_R	size of tuples of relation R in bytes	100-200
$\lvert\lvert R\rvert\rvert$	size of relation R in tuples	
$\lvert R\rvert$	$= \left\lceil \frac{\lvert\lvert R\rvert\rvert \times ts_R}{bs} \right\rceil$, size of R in blocks	

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

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

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

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

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

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

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

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

CPU time to probe a join:

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

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

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

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

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

Fig. 2 Cost Functions

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 1 gives further notation and some basic cost values taken from literature. In Figure 2, we present the cost functions for single operations as we will use them in the remainder of this paper.

5.1 DEFERRED PROBE

The naive way to combine build and probe phase is to execute them one after the other as follows: $\mathsf{Alloc}(H_1)$; $\mathsf{Build}(R_1)$; ... ; $\mathsf{Alloc}(H_N)$; $\mathsf{Build}(R_N)$; $\mathsf{Probe}(I_1..I_N)$; $\mathsf{Free}(H_1)$; ... ; $\mathsf{Free}(H_N)$. We call this *deferred probe*. The total execution time is (cf. Fig. 2 and Tab. 1 for details):

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

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:

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

However, in most environments the build phase is I/O-bound while the probe phase is CPU-bound, i.e.

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

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 free, while during the probe phase, I/O capacities are left free. Thus, the execution time is not optimal:

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

Figures 3 and 4 depict CPU and I/O load of deferred probe evaluating a pipelining 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 (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. 5).

5.2 FLOATING PROBE

To overcome the shortcomings of deferred probe, our approach is to let the build phase and the probe phase overlap. As opposed to deferred probe, this results in

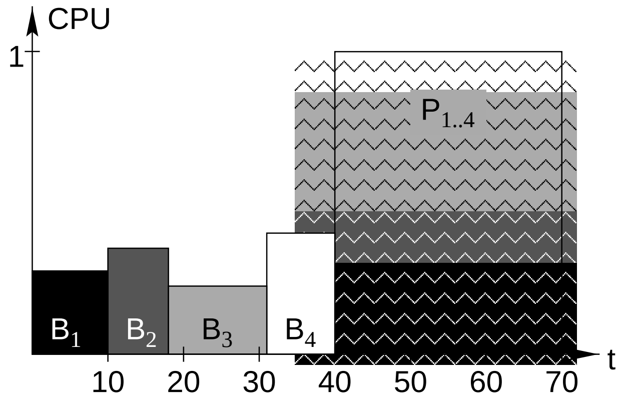

Fig. 3 Sample CPU load (deferred probe)

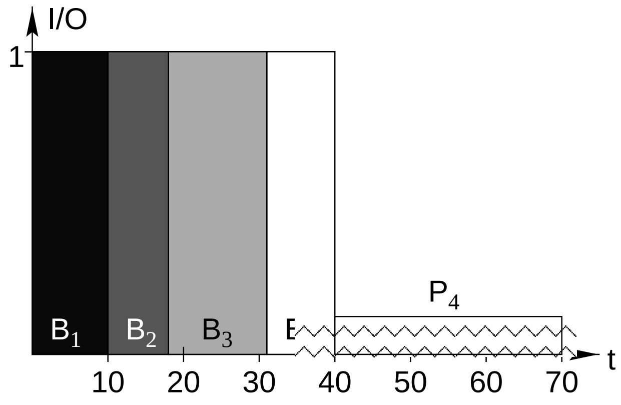

Fig. 4 Sample I/O load (deferred probe)

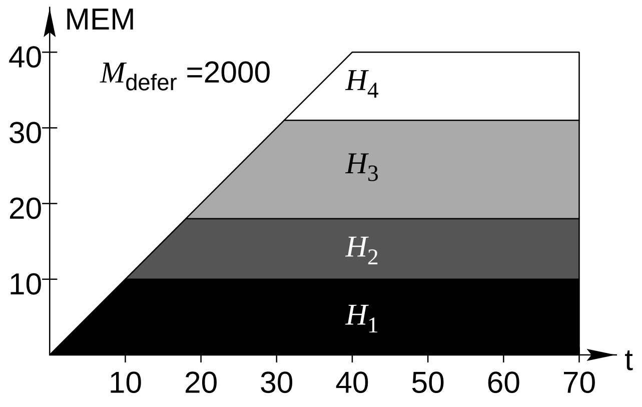

Fig. 5 Sample memory usage (deferred probe)

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, that $\mathsf{Probe}(I_i)$ 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

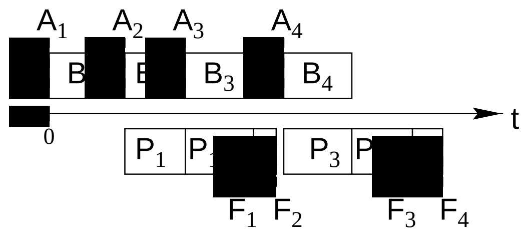

Fig. 6 Sample schedule floating probe

of the last hash table has finished. As building the hash tables is I/O-bound, the elapsed time until all hash tables are build cannot be reduced. But the probe work that has to be done after the last build is reduced, and thus, the overall execution time is reduced.

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

$\mathsf{Probe}(I_1)$ is CPU-bound. In this case, floating probe proceeds as follows (cf. Fig. 6 for a sample schedule): At the beginning, H_1 is built ($\mathsf{Build}(R_1)$). Thereafter, $\mathsf{Probe}(I_1)$ and $\mathsf{Build}(R_2)$ are started simultaneously and executed concurrently. As the output tuples produced by $\mathsf{Probe}(I_1)$ cannot yet be processed by $\mathsf{Probe}(I_2)$, they have to be buffered. To avoid intermediate I/O, this is done in memory. If $\mathsf{Probe}(I_1)$ ends before $\mathsf{Build}(R_2)$, H_1 is deleted. Otherwise, as soon as $\mathsf{Build}(R_2)$ has finished, $\mathsf{Build}(R_3)$ is started and the probe is extended, so that the remaining tuples of I_1 are piped through both probes ($\mathsf{Probe}(I_1..I_2)$). As before, the output of $\mathsf{Probe}(I_1..I_2)$ is buffered in memory. If then $\mathsf{Probe}(I_1..I_2)$ ends before $\mathsf{Build}(R_3)$, H_1 is deleted and the part of I_2 buffered in memory during $\mathsf{Build}(R_2)$ is processed through $\mathsf{Probe}(I_2)$. Otherwise, the probe is extended to $\mathsf{Probe}(I_1..I_3)$, as soon as $\mathsf{Build}(R_3)$ is done. This proceeds until 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 $\mathsf{Probe}(I_i..I_N)$ is executed.

In floating probe, the pipelining segment is dynamically extended to the next join, as soon as its hash table is built. Thus, allocated memory is used as soon as possible. On the other hand, hash tables are deleted as soon as the respective probe is done. Thus, allocated memory is released as soon as it is no longer needed.

Figures 7 and 8 depict CPU and I/O load of floating probe evaluating a pipelining segment with four joins (I_1 is receive via the network and I_5 is written to disk) and Figure 9shows the respective memory usage.

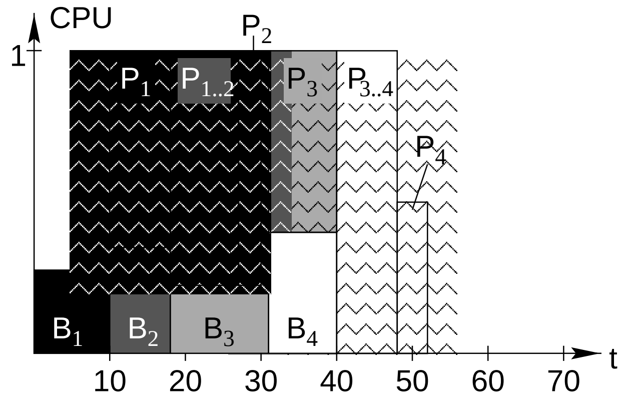

Fig. 7 Sample CPU load (floating probe)

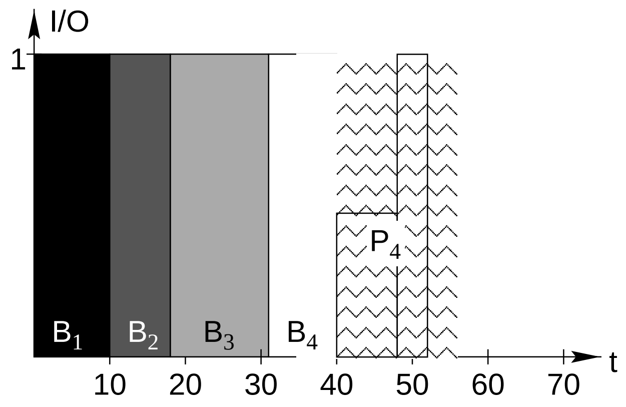

Fig. 8 Sample I/O load (floating probe)

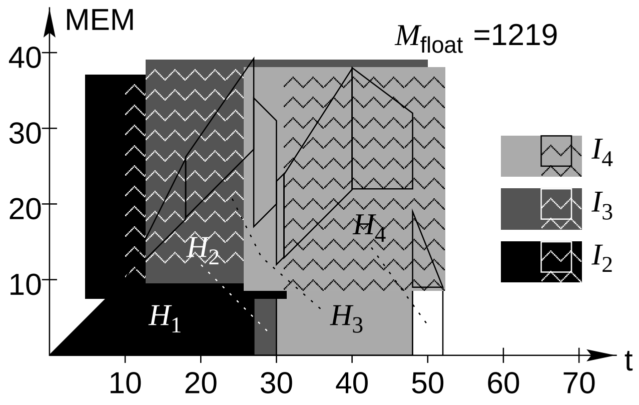

Fig. 9 Sample memory usage (floating probe)

$\mathsf{Probe}(I_1)$ is I/O-bound. Now, we consider the case, that reading I_1 from disk is slower than performing the probe. As I_1 is also full declustered across all disks, there is no sense in running $\mathsf{Probe}(I_1)$ and $\mathsf{Build}(R_2)$ in parallel due to disk access contention. We examined two strategies, how to proceed in this case.

The first is to defer $\mathsf{Probe}(I_1)$ until enough (g) hash tables are built, such that executing $\mathsf{Build}(R_{g+1})$

and $\mathsf{Probe}(I_1..I_g)$ concurrently is (approximately) CPU-I/O-balanced, or at least such that executing $\mathsf{Probe}(I_1..I_g)$ is CPU-bound. Thus, running $\mathsf{Probe}(I_1)$ I/O-bound is avoided. But on the other hand, the start of probing is deferred and $\mathsf{Build}(R_2)$ through $\mathsf{Build}(R_g)$ are run I/O-bound. As soon as $\mathsf{Probe}(I_1..I_g)$ is started, execution continues as usual. We call this strategy *late probing.*

The second strategy is to execute $\mathsf{Probe}(I_1)$ right after $\mathsf{Build}(R_1)$, materializing I_2 completely in memory, and to defer $\mathsf{Build}(R_2)$ until $\mathsf{Probe}(I_1)$ is done. As soon as $\mathsf{Build}(R_2)$ has finished, processing proceeds as usual starting $\mathsf{Build}(R_3)$ and $\mathsf{Probe}(i_2)$ simultaneously. Thus, $\mathsf{Probe}(I_1)$ is run I/O-bound as well as $\mathsf{Build}(R_2)$ thereafter. But on the other hand, probing is started as soon as possible. We call this strategy *early probing.*

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

The first advantage of floating probe is that the overall execution time is reduced as some of the probe work is done before $\mathsf{Build}(R_N)$ has finished. In our example, deferred probe needs 70 units, whereas floating probe needs only 52 units (cf. Figs. 3,4,7,8). Of course, 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

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

Further, the execution time of floating probe cannot be less than half the execution time of deferred probe:

$$T^{\mathsf{min}}_{\mathsf{float}} \geq \max\{O_s(R_1..R_N), C_{Px}(I_1..I_N)\} \geq \frac{T_{\mathsf{defer}}}{2}. \quad (2)$$

The second advantage of floating probe is reduced memory consumption. If any probe finishes before $\mathsf{Build}(R_N)$ is done, the corresponding hash table is released, and thus, the memory usage area (i.e. amount of memory used $\times$ time for that it is occupied) 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. 5 & 9).

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 minimal execution time of floating probe is:

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

6 ANALYSIS

According to the presentation of floating probe in the previous section, it seems to be rather complicated do 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.

6.1 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 in distinct 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 the 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 1. According to the aforementioned strategy, the simulator assumes distinct build and probe threads, one of each per processor.

6.2 EXPERIMENTS

We randomly generated pipelining segments of several classes. Each class is characterized by the length $N \in \{4, 8, 16\}$ of the pipelining segment and the location L of I_1 and I_{N+1}. Due to space limits, we restrict our discussion here to the two cases that either (1) I_1 is initially stored on disk and I_{N+1} finally has to be stored on disk ($L = \texttt{disk}$), or that (2) I_1 is received via network and I_{N+1} is sent to the network ($L = \texttt{net}$). 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 (1).

For each segment $S_i^{L,N}$, we simulated the execution with both deferred probe and floating probe[1] for different degrees of parallelism ($p \in \{1, 2, 4, 8\}$, $d = p$). To compare the performance of deferred probe and floating probe, we calculated the relative execution time $T_{\mathsf{float}}(S_i^{L,N}, p)/T_{\mathsf{defer}}(S_i^{L,N}, p)$. Within each class we calculated the average relative execution time over all the $n = 360$ queries:

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

Figures 10 and 11 show the average relative execution times with ($L = \texttt{disk}$) and without probe-I/O ($L = \texttt{net}$), respectively. Floating probe outperforms deferred probe in any case ($\overline{T}_{\mathsf{f/d}}(L, N, p) < 1$), and the improvement increases with the length of the pipelining segment. 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 $L = \texttt{disk}$ and up to 31% of execution time for $L = \texttt{net}$. Remember, that at most 50% can be saved (cf. (2)). The average saving amounts to approximately 16% for $L = \texttt{disk}$ and 24% for $L = \texttt{net}$.

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 $M(S_i^{L,N}, p)$. Analogous to the average relative execution time, we calculated the average relative memory usage $\overline{M}_{\mathsf{f/d}}(L, N, p)$. Figures 12 and 13 show the results with ($L = \texttt{disk}$) and without probe-I/O ($L = \texttt{net}$), respectively. Again, floating probe performs better—i.e. needs less memory—than deferred probe. Here, the differences between $L = \texttt{disk}$ and $L = \texttt{net}$ are negligible. Floating probe saves up to 80% (55% on average) of memory allocation compared to deferred probe.

[1]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.

7 CONCLUSION

This paper addresses the topic of efficient resource utilization during query execution in parallel database systems. We presented *floating probe*, a new technique to evaluate pipelining segments in shared-everything environments. Floating probe balances the CPU- and I/O-workload between the I/O-bound build phase and the CPU-bound probe phase of pipelining segments as good as possible with respect to the data dependencies between both phases. Thus, floating probe achieves better resource utilization than conventional deferred probe. This in turn leads to further advantages of floating probe compared to deferred probe: (1) Floating probe provides shorter execution times while (2) consuming less memory than deferred probe. Floating probe achieves these improvements without explicit scheduling, thus, floating probe neither needs any preliminary cost estimations nor does it cause any scheduling overhead.

We used various simulation experiments to compare floating probe and deferred probe in detail. The results show, that floating probe outperforms deferred probe in any case in terms of execution time and memory usage.

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.

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.

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.

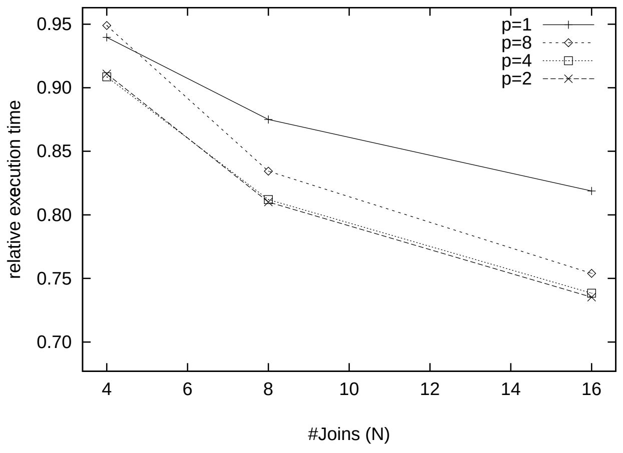

Fig. 10 $\overline{T}_{\mathtt{f/d}}(\mathtt{disk}, N, p)$

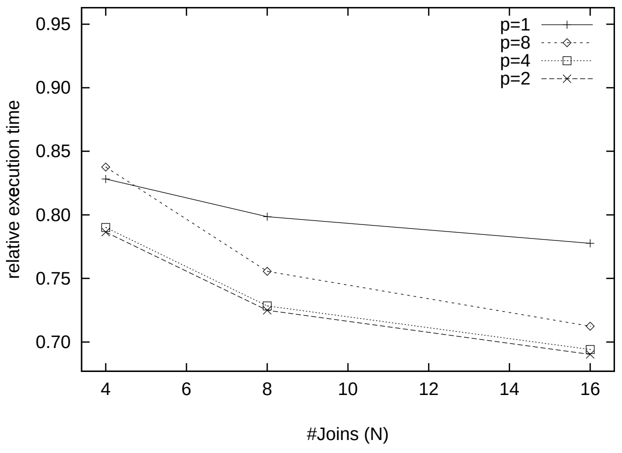

Fig. 11 $\overline{T}_{\mathtt{f/d}}(\mathtt{net}, N, p)$

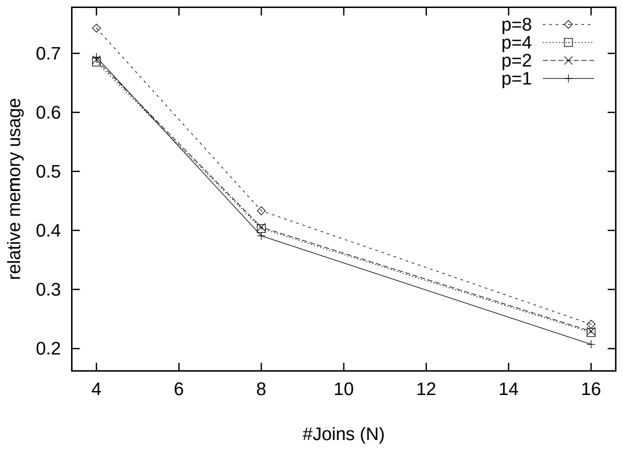

Fig. 12 $\overline{M}_{\mathtt{f/d}}(\mathtt{disk}, N, p)$

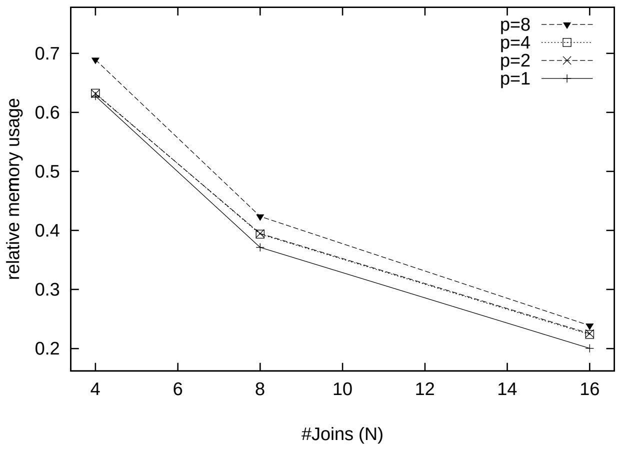

Fig. 13 $\overline{M}_{\mathtt{f/d}}(\mathtt{net}, N, p)$

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. and Obermaier, J. K. (1997). Efficient Resource Utilization in Shared-Everything Environments. Technical Report INS-R9711, CWI, Amsterdam, The Netherlands.

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.

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.