Efficient Resource Utilization in Shared-Everything Environments

S. Manegold, J.K. Obermaier

Information Systems (INS)

INS-R9711 December 31, 1997

Report INS-R9711
ISSN 1386-3681

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

Efficient Resource Utilization in Shared-Everything Environments

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

[1] *CWI, P.O.Box 94079, 1090 GB Amsterdam, The Netherlands*
manegold@cwi.nl

[2] *ABB Corporate Research Ltd, Computer Engineering Dept., Information Technology CHCRC.C2*
5405 Baden, 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.

1991 Computing Reviews Classification System: [H.2.4] Parallel Database Systems, Query Processing
Keywords and Phrases: parallel and distributed databases, parallel query processing, dynamic load balancing, efficient resource utilization
Note: This work was partly supported by the German Research Council under contract DFG Fr 1142/1-1.

1. Introduction

Parallel processing in database systems is a key to the required performance improvements of modern database applications. The increasing complexity of queries and the increasing transaction throughput over a growing amount of data require higher performance of relational database systems. Hence, more and more commercial database vendors are integrating parallelism in their products [DG92, Gra95].

In general, parallelism for the evaluation of database queries is classified into three main categories [SD90, YCWT93]: *inter-query*, *inter-operator*, and *intra-operator parallelism*. Much research has been performed to determine which kind of parallelism to use for query processing. Database machines research concentrated on intra-operator parallelism. Most commercial database systems have focused on inter-query parallelism, so far [Gra95, Val93]. Recently, the use of inter-operator parallelism has been investigated [CLYY92, HS93, SD90, SYT93, SE93, WA91, ZZBS93]. Pipelining parallelism is of particular interest. In [SD90] Schneider and DeWitt 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 [CLYY92]. 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.

In [SYT93] this idea is expanded to bushy operator trees. 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.

A major problem with the usage of pipelining parallelism are dependencies between operators, i.e. the performance of the pipelining execution is dominated by the slowest operator. Hence, it is important to predict the workload of the operators precisely to decide its degree of parallelism. There are two problems: failures in the prediction of the operators' work (*execution skew*) and *discretization error* [SE93, WFA95], i.e. a fixed number of processors cannot be assigned to the operators such that every operator reaches its optimal degree of parallelism. Minimizing discretization error by using more processes than processors as a straightforward solution adds the overhead of process context switching. An additional problem with the dependencies between operators are the *startup* and *shutdown execution delay* [GHK92, WA91, WFA95]. Processors assigned to operators at the end of a pipeline are idle at the beginning of the computation, whereas processors assigned to operators at the begin of the pipeline are idle towards the end of the execution. In [MOW97], we presented DTE, a new strategy to execute the probe phase of pipelining segments in shared-everything environments, that avoids both, discretization error and startup/shutdown delay, and is resistant against execution skew. Thus, DTE 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 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 [Hon92]. This algorithm is restricted to scheduling distinct data-independent task (i.e. pipelining segments), whereas we focus on executing the two data-dependent phases of a single 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. [CLYY92, SD90, SYT93]). To achieve this, the optimizer splits a sequence of non-blocking operators into multiple segments if necessary.

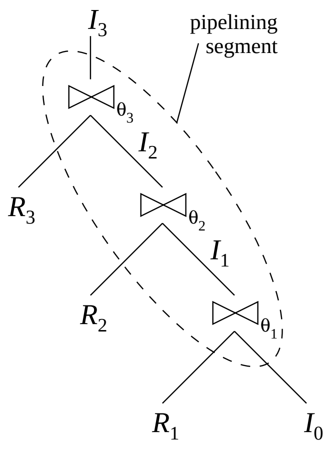


Figure 1: A pipelining segment

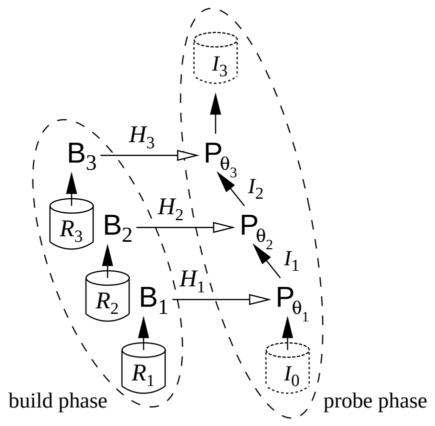


Figure 2: Build phase and probe phase

Figure 1 depicts a sample pipelining segment consisting of three joins. R_i, I_i, and θ_i denote the inner input relation, the intermediate result, and the join predicate of the i-th join, respectively. I_0 denotes the outer input relation of the first join (i.e. the outer input of the whole segment). Each input relation is either a base relation or the result of an other pipelining segment. All R_i are materialized on disk. I_0 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 [SYT93]. Evaluation of a segment proceeds in two phases: The first phase loads all inner relations of joins in the segment by parallel I/O and builds the (hash) indices in parallel (*build phase*). The second phase pipes all tuples of the outer relation through selections, projections, or probe phases of joins (*probe phase*).

Figure 2 depicts the build phase an the probe phase of the sample segment. B_i denotes the operation to build the hash table H_i of the i-th join, while P_{θ_i} denotes the operation to probe I_{i-1} against H_i using the predicate θ_i.

In the following two sections, we present our execution strategies for the build phase and the probe phase, respectively. Thereafter, in Section 5, we show, that executing these two phases one after another is not optimal due to inefficient resource utilization. Then, we present floating probe, our new strategy to combine both phases, that provides shorter execution times while using less memory.

3. Table Building Phase

In this section, we discuss execution strategies for the build phase of pipelining segments. First, we present a strategy how to build a single hash table in parallel. Then, we study techniques to build all the hash tables of one pipelining segment.

3.1 Building a single hash table in parallel

Shared-everything systems provide uniform and parallel access to all disks. We assume that all base relations are 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.

We use the following strategy to build a single hash table in parallel, i.e. using all disks and all CPUs. One thread per CPU is started. Each thread reads tuples (one at a time) from a shared buffer pool, calculates the hash value and inserts the tuple into the corresponding hash table in shared memory. As with DTE (cf. Sec. 4), the strategy provides optimal load balancing. The shared buffer pool is continuously fed with pages of the base relation that are read from disk. To do this, we extend one of the aforementioned threads by the functionality to invoke 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 page from disk, this single I/O thread 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.

3.2 Building multiple hash tables

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 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. Remember, that we assume full declustering of all base relations. Thus, 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 here.

4. Tuple Probing Phase

The key idea of our strategy to evaluate the probe phase of pipelining segments is to dynamically assign the available processors to the data that must be processed. We do this by gathering all operators of a pipelining segment into one stage and assigning all processors to this stage. This leads to optimal load balancing and efficient resource utilization without 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, we assign one thread per processor. Each thread is able to perform all operations within the active pipelining segment. In [MOW97], we present this strategy — called *Data Threaded Execution (DTE)* — in detail. In the reminder of this section, we give an overview of DTE.

Evaluation of a pipelining segment proceeds as follows: The input tuples for the pipelining segment are provided in a single 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. Figure 3 depicts the data threaded execution of a pipelining 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. But this time is at most the time that one processor need to process one tuple though the pipelining segment.

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 processors are working as long as there are input tuples in the queue. i.e. neither startup delay nor discretization error occur with DTE.

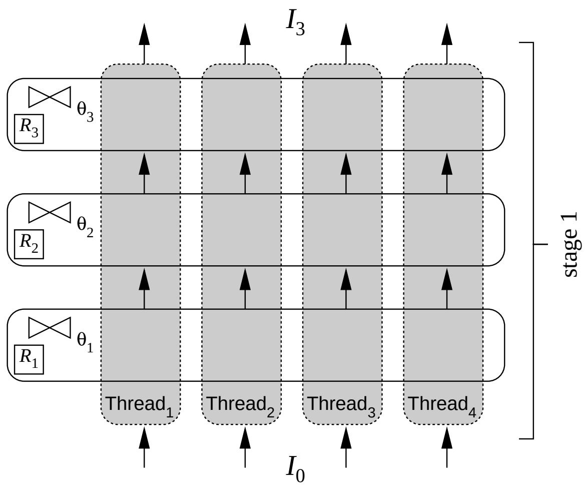


Figure 3: DTE

In particular, 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. Of cause, when such errors lead to a non-optimal query tree, DTE cannot compensate this error. But it still provides a stable execution in the sense of efficient resource utilization without overhead.

5. Building and Probing

Before we discuss the different strategies how to combine build phase and probe phase, we introduce the notation we use in the remainder of this paper. Build(R_i) (in figures abbreviated by B$_i$) denotes the building of the i-th hash table. This includes reading R_i from disk. Alloc(H_i) (A$_i$) denotes the allocation of the memory that is needed for a hash table. It returns `yes` if the allocation was successful, i.e. if there was enough memory available, and `no` otherwise. Releasing the respective amount of memory is denoted by Free(H_i) (F$_i$). Probe(I_{i-1}) (P$_{i-1}$) denotes the probing of intermediate result I_{i-1} through i-th join within the pipeline using DTE. Probe($I_{i-1}..I_{j-1}$) (P$_{i-1..j-1}$) denotes the parallel probing of the joins i through j ($1 \leq i \leq j \leq N$) using DTE. Thus both, Probe(I_{i-1}) and Probe($I_{i-1}..I_{j-1}$) represent the execution of the respective subset of operators of the whole pipeline (Probe($I_0..I_{N-1}$)). Table 1 gives further notation and some basic cost values taken from litrature. In Figure 4, we present the cost functions for single operations as we will use them in the remainder of this paper.

5.1 Deferred Probe

The simplest way to combine build and probe phase is the classical one: 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 pipelining segment (i.e. build and probe phase) proceeds as follows: Alloc(H_1); Build(R_1); ..; Alloc(H_N); Build(R_N); Probe($I_0..I_{N-1}$); Free(H_1); ..; Free(H_N). The total execution time is (cf. Fig. 4 and Tab. 1 for details):

$$T'_{\mathsf{deferred}} = \sum_{i=1}^{N} \max\{O_s(R_i)\ , C_B(R_i)\} + \max\{O_r(I_0) + O_r(I_N)\ , C_{Px}(I_0..I_{N-1})\}\,.$$

Suppose that build phase and probe phase either both are I/O-bound

$$\forall i \in \{1,..,N\} : O_s(R_i) > C_B(R_i) \qquad \wedge \qquad O_r(I_0) + O_r(I_N) > C_{Px}(I_0..I_{N-1})$$

name	description	value
N	number of joins	
R_i	base relations, $i \in \{1, \ldots, N\}$	
H_i	corresponding hash tables, $i \in \{1, \ldots, N\}$	
I_i	intermediate results, $i \in \{0, \ldots, N\}$	
p	number of CPUs	
d	number of disks	
T_M	CPU-time to transfer one tuple between CPU and memory	10.0 μs
T_B	CPU-time per tuple to build a hash table	5.5 μs
T_P	CPU-time to probe one tuple against a hash table	4.0 μs
T_G	CPU-time to generate one result tuple of a join	30.0 μs
T_I	CPU-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 Bytes
$\|\|R\|\|$	size of relation R in tuples	
$\|R\|$	$= \left\lceil \frac{\|\|R\|\| \times ts_R}{bs} \right\rceil$, size of relation R in blocks	

Table 1: Notation

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

$$O_s(R_i) = T_S + \left\lceil \frac{|R_i|}{d} \right\rceil (T_W + T_R) \; ; \qquad O_s(R_i..R_j) = \sum_{k=i}^{j} O_s(R_k)$$

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

$$O_r(R_i) = \left\lceil \frac{|R_i|}{d} \right\rceil (T_S + T_W + T_R) \; ; \qquad O_r(R_i..R_j) = \sum_{k=i}^{j} O_r(R_k)$$

CPU time per relation to init I/O and to transfer a relation between CPU and memory:

$$C_x(R_i) = \left\lceil \frac{|R_i|}{p} \right\rceil T_I + \left\lceil \frac{||R_i||}{p} \right\rceil T_M \; ; \qquad C_x(R_i..R_j) = \sum_{k=i}^{j} C_x(R_k)$$

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

$$C_B(R_i) = \left\lceil \frac{|R_i|}{p} \right\rceil T_I + \left\lceil \frac{||R_i||}{p} \right\rceil T_B \; ; \qquad C_B(R_i..R_j) = \sum_{k=i}^{j} C_B(R_k)$$

CPU time to probe a join:

$$C_P(I_i) = \left\lceil \frac{||I_i||}{p} \right\rceil T_P + \left\lceil \frac{||I_{i+1}||}{p} \right\rceil T_G \; ; \qquad C_P(I_i..I_j) = \sum_{k=i}^{j} C_P(I_k)$$

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

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

Figure 4: Cost Functions

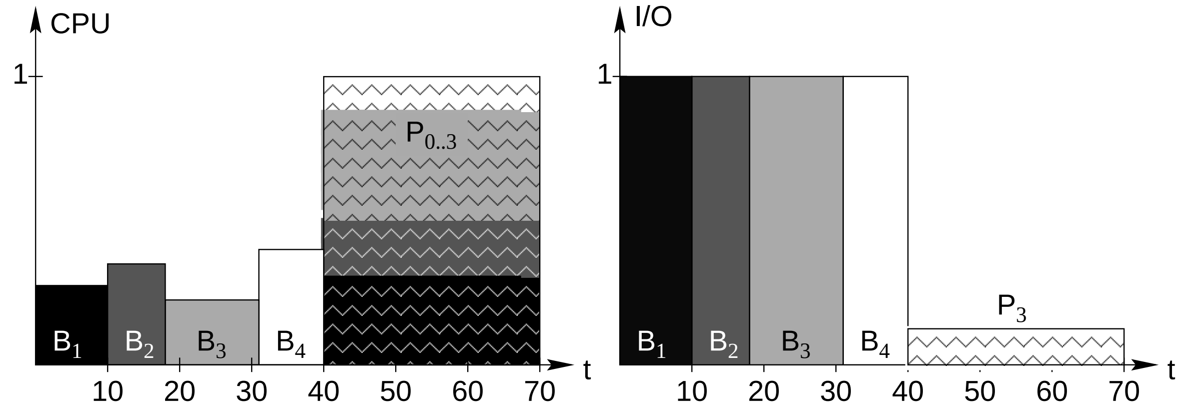


Figure 5: Sample CPU and I/O load (deferred probe)

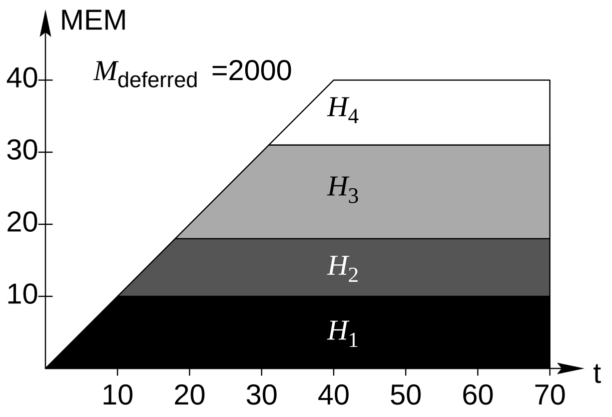


Figure 6: Sample memory usage (deferred probe)

or both are CPU-bound

$$\forall i \in \{1,..,N\} : O_s(R_i) \;<\; C_B(R_i) \qquad \wedge \qquad O_r(I_0) + O_r(I_N) \;<\; C_{Px}(I_0..I_{N-1}),$$

then this execution strategy provides the minimal execution time:

$$T''_{\text{deferred}} \;=\; \max\{O_s(R_1..R_N) + O_r(I_0) + O_r(I_N)\ ,\ C_B(R_1..R_N) + C_{Px}(I_0..I_{N-1})\}.$$

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.

$$\forall i \in \{1,..,N\} : O_s(R_i) \;>\; C_B(R_i) \qquad \wedge \qquad O_r(I_0) + O_r(I_N) \;<\; C_{Px}(I_0..I_{N-1}). \tag{5.1}$$

In this case, the strategy presented above 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:

$$T_{\text{deferred}} \;=\; O_s(R_1..R_N) + C_{Px}(I_0..I_{N-1}) \;>\; T''_{\text{deferred}}.$$

Figure 5 depicts CPU and I/O load during the evaluation of a pipelining segment with four joins using deferred probe.

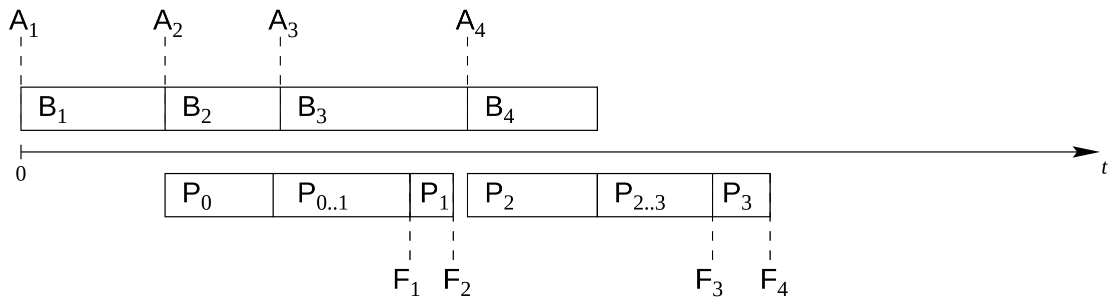


Figure 7: Sample schedule floating probe

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 this strategy. 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. 6).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 $\times$ time that the memory is occupied).

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 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.

The point is, that $\mathsf{Probe}(I_{i-1})$ can be started as soon as $\mathsf{Build}(R_i)$ has finished, i.e. $\mathsf{Probe}(I_{i-1})$ can be executed in parallel with $\mathsf{Build}(R_{i+1})$. Thus, compared to conventional pipelining, some of the probe work is done before the build 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 $\mathsf{Probe}(I_0)$ is CPU-bound (I_0 already resides in memory, is received via a fast network, or even reading from disk is faster than performing the probing), or $\mathsf{Probe}(I_0)$ is I/O-bound (reading I_0 from disk is slower than performing the probing).

For the moment, we assume that $\mathsf{Probe}(I_0)$ is CPU-bound. Then, the strategy proceeds as follows (cf. Fig. 7 for a sample schedule of floating probe): At the beginning, the hash table of the first join is built ($\mathsf{Build}(R_1)$). Thereafter, $\mathsf{Probe}(I_0)$ and $\mathsf{Build}(R_2)$ are started simultaneously and executed concurrently. As the output tuples that $\mathsf{Probe}(I_0)$ produces cannot immediately be processed by $\mathsf{Probe}(I_1)$, they have to be buffered. To avoid intermediate I/O, this should be done in memory. If $\mathsf{Probe}(I_0)$ ends before $\mathsf{Build}(R_2)$ is ready, the hash table of the first join (H_1) can be 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_0 are piped through both probes ($\mathsf{Probe}(I_0..I_1)$). As before, the output of $\mathsf{Probe}(I_0..I_1)$ is materialized in memory. If then $\mathsf{Probe}(I_0..I_1)$ ends before $\mathsf{Build}(R_3)$ is ready, H_1 can be deleted and the part of I_1 that was materialized in memory during $\mathsf{Build}(R_2)$ is processed through $\mathsf{Probe}(I_1)$. Otherwise, the probe is extended to the third join ($\mathsf{Probe}(I_0..I_2)$), as soon as $\mathsf{Build}(R_3)$ is done. This proceeds until the last hash table is built. After that, only probing is done until all tuples are processed: For each I_i that is (partially) materialized in memory $\mathsf{Probe}(I_{i-1}..I_{N-1})$ is executed.

In our new strategy, 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 phase is done. Thus, allocated memory is released as soon as it is no longer needed.

We call this strategy *floating probe*. Figure 8 presents floating probe as pseudo code[1]. Figure 10

```
begin
    Init();
    do
        if $next \le N$ then
            if $first \le last$ 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 8: Floating probe (CPU-bound Probe(I_0))

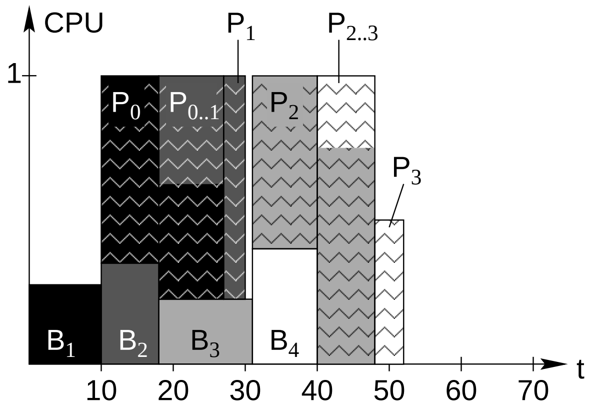
```
Init();
do
    BuildOnly($R_{next}$);
until ProbeOnly($I_{first}..I_{last}$) is I/O-bound;
```

a) Replacement for Init() in Fig. 8: *late probing*

```
Init();
BuildOnly($R_{next}$);
ProbeOnly($I_{first}..I_{last}$);
```

b) Replacement for Init() in Fig. 8: *early probing*

Figure 9: Floating probe (I/O-bound Probe(I_0))



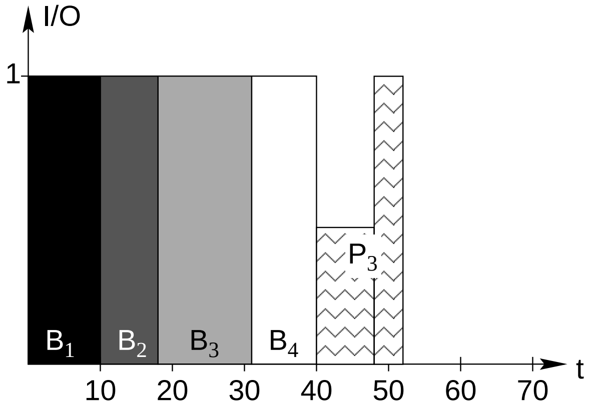


Figure 10: Sample CPU and I/O load (floating probe)

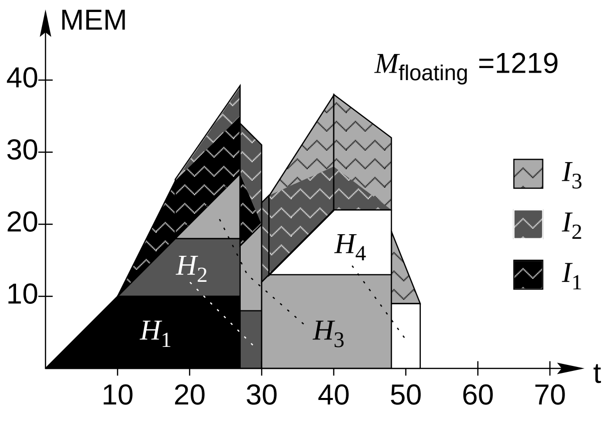


Figure 11: Sample memory usage (floating probe)

depicts CPU and I/O load during the evaluation of a pipelining segment with four joins using floating probe (I_0 is receive via the network and I_4 is written to disk) and Figure 11 shows the respective memory usage.

Now, let's consider the case, that $\mathsf{Probe}(I_0)$ is I/O-bound, i.e. reading I_0 from disk is slower than performing the probe. As I_0 is also full declustered across all disks, there is no sense in running $\mathsf{Probe}(I_0)$ 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 $\mathsf{Probe}(I_0)$ until enough (g) hash tables are built, such that executing $\mathsf{Build}(R_{g+1})$ and $\mathsf{Probe}(I_0..I_{g-1})$ concurrently is (approximately) CPU-I/O-balanced, or at least such that executing $\mathsf{Probe}(I_0..I_{g-1})$ is CPU-bound. Thus, running $\mathsf{Probe}(I_0)$ 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 is started $\mathsf{Probe}(I_0..I_{g-1})$, execution continues as usual. We call this strategy *late probing* (cf. Fig. 9a).

The second strategy is to execute $\mathsf{Probe}(I_0)$ right after $\mathsf{Build}(R_1)$, materializing I_1 completely in memory, and to defer $\mathsf{Build}(R_2)$ until $\mathsf{Probe}(I_0)$ is done. Thus, $\mathsf{Probe}(I_0)$ 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* (cf. Fig. 9b).

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. $\mathsf{Probe}(I_{N-1})$ can only be processed after $\mathsf{Build}(R_N)$ is done. Hence, there is no I/O interference.

The gain of our new strategy is twofold: On the one hand 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. 5 & 10). 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 given by

$$\begin{aligned} T^{\min}_{\mathsf{floating}} &= \max\{O_s(R_1..R_N) + O_s(I_0) + O_s(I_N)\ ,\ C_B(R_1..R_N) + C_{Px}(I_0..I_{N-1})\} \\ &\overset{(5.1)}{\leq} \max\{O_s(R_1..R_N) + C_{Px}(I_0..I_{N-1})\ ,\ O_s(R_1..R_N) + C_{Px}(I_0..I_{N-1})\} \\ &= T_{\mathsf{deferred}}. \end{aligned}$$

The following equation shows, that the execution time of floating probe cannot be less than 50% the execution time of deferred probe:

$$T^{\min}_{\mathsf{floating}} \geq \max\{O_s(R_1..R_N) + O_s(I_0)\ ,\ C_{Px}(I_0..I_{N-1})\} \geq \frac{T_{\mathsf{deferred}}}{2}. \tag{5.2}$$

On the other hand, if any probes finish before $\mathsf{Build}(R_N)$ is done, the corresponding hash tables can be released, and thus, the memory usage area (i.e. amount of memory used $\times$ time for that it is occupied) are smaller than that of the classical strategy. In our example, the memory usage area of deferred probe amounts to 2000 units, whereas that of floating probe is only 1219 units (cf. Figs. 6 & 11).

A drawback of this strategy 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

[1]The Procedures that are used here and with the pseudo codes of the following strategies are presented in Figure 16 in the Appendix.

— for the moment, the minimal execution time of our new strategy is:

$$\begin{aligned} T_{\mathsf{floating}} &= \max\{O_s(R_1)\ ,\ C_B(R_1)\} + \\ &\quad \max\{O_s(R_2..R_N) + O_s(I_0)\ ,\ C_B(R_2..R_N) + C_x(I_0) + C_P(I_1..I_{N-2})\} + \\ &\quad \max\{O_s(I_N)\ ,\ C_P(I_N) + C_x(I_N)\} \\ &\geq O_s(R_1) + \\ &\quad \max\{O_s(R_2..R_N) + O_s(I_0)\ ,\ C_B(R_2..R_N) + C_x(I_0) + C_P(I_1..I_{N-2})\} + \\ &\quad C_P(I_N) + C_x(I_N). \end{aligned}$$

6. Analysis

6.1 Simulation Model

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.

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 the conventional strategy, we designed and implemented an event driven simulator using the Sim++ package [Fis95]. 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_0 and I_N. Due to space limits, we restrict our discussion here to the two cases that either (1) I_0 is initially stored on disk and I_N finally has to be stored on disk ($L = \texttt{disk}$), or that (2) I_0 is received via network and I_N 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 $n = 360$ different segments for each class, where the tuple size of each base relation and each intermediate result varied between 100 and 200 bytes and the size of each base relation and each intermediate result varied between 10^3 and $2 \cdot 10^5$ tuples. To guarantee, that the build phase is I/O-bound while the probe phase is CPU-bound, all segments fulfilled condition (5.1).

For each segment $S_i^{N,L}$, we simulated the execution with both strategies, deferred probe and floating probe[2], 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 $T_{\mathsf{floating}}(S_i^{N,L}, p)/T_{\mathsf{deferred}}(S_i^{N,L}, p)$. Within each class—identified by N, p, and L—we calculated the

[2]If I_0 and I_N 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.

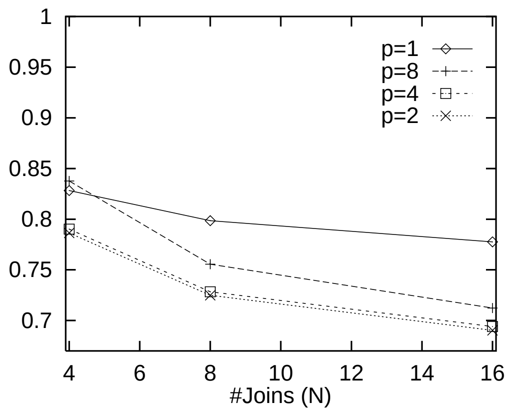


Figure 12: $\overline{T}_{\mathsf{f/d}}^{N,\mathtt{net},p}$

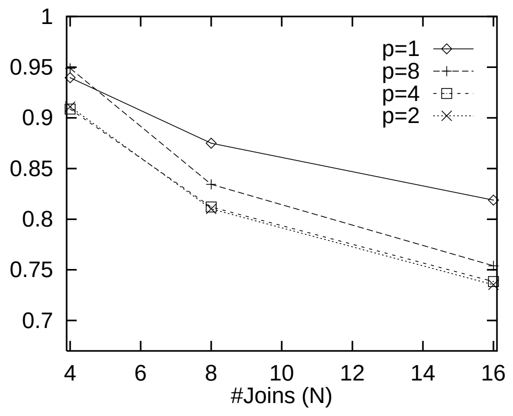


Figure 13: $\overline{T}_{\mathsf{f/d}}^{N,\mathtt{disk},p}$

average relative execution time over all the 360 queries:

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

Figures 12 and 13 show the average relative execution times with and without probe-I/O, respectively. Floating probe outperforms deferred probe in any case ($\overline{T}_{\mathsf{f/d}}^{N,L,p} < 1$), 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 31% of execution time for $L = \mathtt{net}$ and up to 27% for $L = \mathtt{disk}$. Remember, that at most 50% can be saved (cf. (5.2)). The average saving amounts to approximately 24% for $L = \mathtt{net}$ and 16% for $L = \mathtt{disk}$.

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 $M(S_i^{N,L},p)$. Analogous to the average relative execution time, we calculated the average relative memory usage $\overline{M}_{\mathsf{f/d}}^{N,L,p}$. Figures 14 and 15 show the results with and without probe-I/O, respectively. Again, floating probe performs better—i.e. needs less memory—than deferred probe. Here, the differences between $L = \mathtt{net}$ and $L = \mathtt{disk}$ are negligible. Floating probe saves up to 80% (55% on average) of memory allocation compared to deferred probe.

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.

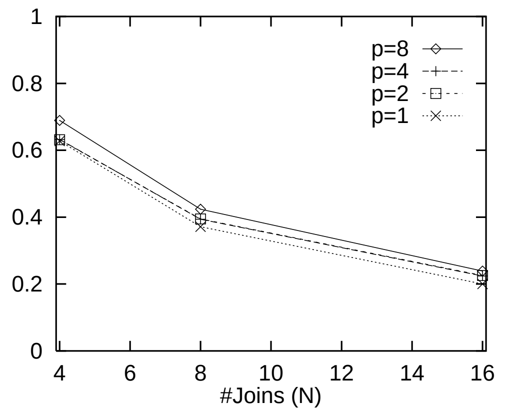


Figure 14: $\overline{M}_{\texttt{f/d}}^{N,\texttt{net},p}$

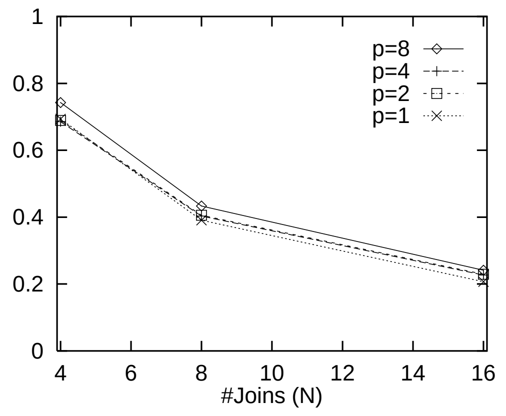


Figure 15: $\overline{M}_{\texttt{f/d}}^{N,\texttt{disk},p}$

The results show, that floating probe outperforms deferred probe in any case in terms of execution time and memory usage.

Now, we plan to implement floating probe on different platforms (e.g. on multi-processor SMP workstations and a Cray T3E) to validate our simulation results. Further, we plan to use floating probe as a building block to build a query evaluation system for hybrid architectures.

References

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

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

[Fis95] P. A. Fishwick. *Simulation Model Design and Execution.* Prentice Hall, Englewood Cliffs, NJ, USA, 1995.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query Optimization for Parallel Execution. In *Proc. ACM SIGMOD Int'l. Conf.*, San Diego, CA, USA, June 1992.

[Gra95] J. Gray. A Survey of Parallel Database Techniques and Systems. In *Tutorial Handouts of the 21st Int'l. Conf. on Very Large Data Bases*, Zurich, Switzerland, September 1995.

[Hon92] W. Hong. Exploiting Inter-Operation Parallelism in XPRS. In *Proc. ACM SIGMOD Int'l. Conf.*, San Diego, CA, USA, June 1992.

[HS93] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. *Distr. and Parallel Databases*, 1(1), 1993.

[MOW97] S. Manegold, J. K. Obermaier, and F. Waas. Load Balanced Query Evaluation in Shared-Everything Environments. In *Proc. European Conf. in Parallel Processing*, Passau, Germany, August 1997.

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

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

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

[Val93] P. Valduriez. Parallel Database Systems: Open Problems and New Issues. *Distr. and Parallel Databases*, 1(2), 1993.

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

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

[YCWT93] P. S. Yu, M.-S. Chen, J. L. Wolf, and J. Turek. Parallel Query Processing. In N. R. Adam and B. K. Bhargava, editors, *Advanced Database Systems*, volume 759 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[ZZBS93] M. Ziane, M. Zait, and P. Borla-Salamet. Parallel Query Processing in DBS3. In *Proc. Int'l. Conf. on Parallel and Distr. Inf. Sys.*, San Diego, CA, USA, January 1993.

Appendix: Procedures

```
procedure Init() do
    $toBuild[1..N]$      := {1,..,1};  // part of each hash table that still has to be built
    $toProbe[0..N-1]$    := {1,..,1};  // part of each intermediate result that still has to be probed
    $allocated[1..N]$    := no;        // memory for hash table already allocated ?
    $next$               := 1;         // next hash table that has to be built
    $first$              := 0;         // first intermediate result that has to be probed
    $last$               := -1;        // last intermediate result that can be probed
    $built$              := 0;         // part of a hash table that has currently been built
    $probed$             := 0;         // part of an intermediate result that has currently 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 $i \in \{first, \ldots, last\}$ do $toProbe[i]$ -= $probed$; od;
    while $toProbe[first] = 0 \wedge first < N$ 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 $i \in \{first, \ldots, last\}$ do $toProbe[i]$ -= $probed$; od;
    while $toProbe[first] = 0 \wedge first < N$ do $first$ ++; Free($H_{first}$); od;
    $toBuild[next]$ -= $built$;
    if $toBuild[next] = 0$ then $next$ ++; $last$ ++; fi;
od;
```

Figure 16: Procedures