ON PARALLEL EXECUTION OF MULTIPLE PIPELINED HASH JOINS

Hui-I Hsiao, Ming-Syan Chen and Philip S. Yu
IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
email: {hhsiao, mschen, psyu}@watson.ibm.com

Abstract

In this paper we study parallel execution of multiple pipelined hash joins. Specifically, we deal with two issues, processor allocation and the use of hash filters, to improve parallel execution of hash joins. We first present a scheme to transform a bushy execution tree to an allocation tree, where each node denotes a pipeline. Then, processors are allocated to the nodes in the allocation tree based on the concept of synchronous execution time such that inner relations (i.e., hash tables) in a pipeline can be made available approximately the same time. In addition, the approach of hash filtering is investigated to further improve the overall performance. Performance studies are conducted via simulation to demonstrate the importance of processor allocation and to evaluate various schemes using hash filters. Simulation results indicate that processor allocation based on the allocation tree significantly outperforms that based on the original bushy tree, and that the effect of hash filtering becomes prominent as the number of relations in a query increases.

1 Introduction

As parallelism is recognized as a powerful and cost-effective means to execute complex database operations, it has become imperative to develop efficient solution procedures to handle multi-join queries in parallel database systems [2] [7] [8] [9] [11] [13] [24]. A query plan is usually compiled into a tree of operators, called a join sequence tree, where a leaf node represents an input relation and an internal node represents the resulting relation from joining the two relations associated with its two child nodes. There are three categories of query trees: left-deep trees, right-deep trees, and bushy trees, where the first two are also called linear execution trees, or sequential join sequences. A significant amount of research efforts has been elaborated upon developing join sequences to improve the query execution time. The work reported in [22] was among the first to explore sequential join sequences, and there have been several results reported for sequential join sequences. Owing to the technology advances, it has recently attracted an increasing amount of attention to explore the use of bushy trees for parallel query processing. A combination of analytical and experimental results was given in [14] to shed some light on the complexity of choosing left-deep and bushy trees. An integrated approach dealing with both intra-operator and inter-operator parallelism was presented in [16], where a greedy scheme taking various join methods and their corresponding costs into consideration was proposed. A heuristic approach that deals with a query plan tree for effective resource assignments in a bottom-up manner was presented in [23]. A two-step approach to handle join sequence scheduling and processor allocation for parallel query processing was devised in [5]. A hierachical approach was proposed in [27] to schedule the execution of multiple queries. In addition, various query plans in processing multi-join queries in a shared-nothing architecture were studied in [19] [21].

Among various join methods, the hash join has been the focus of much research effort and reported to have performance superior to that of others, particularly because it presents an opportunity for pipelining [6] [17] [18] [26]. A pipeline of hash joins is composed of several stages, each of which is associated with one join operation that can be executed, in parallel, by several processors. In each stage, the inner relation is used to build the hash table. Though pipelining has been shown to be very effective in reducing the query execution time, prior studies on pipelined hash joins have focused mainly on heuristic methods for query plan generation. Most of the prior work on query plan generation, such as static right-deep scheduling, dynamic bottom-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

SIGMOD 94- 5/94 Minneapolis, Minnesota, USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50

up scheduling [21], segmented right-deep trees [4], and zigzag trees [28], resorted to simple heuristics to allocate processors to pipeline stages. It has been shown that for sort-merge joins, the execution of bushy trees can outperform that of linear trees, especially when the number of relations in a query is large [5]. However, as far as the hash join is concerned, the scheduling for an execution plan of a bushy tree structure is much more complicated than that of a right-deep tree structure. Particularly, it is very difficult to achieve the synchronization required for the execution of bushy trees such that the effect of pipelining can be fully utilized. This is the very reason that most prior studies on pipelined hash joins focused on the use of right-deep trees.

In this paper we explore two important issues, processor allocation and the use of hash filters, to improve the parallel execution of hash joins. Note that to exploit the opportunity of pipelining for hash join execution, one would naturally like to identify and execute a group of hash joins in the bushy tree in a way of pipelining. However, it can be seen that such regrouping of joins, while taking advantage of pipelining, makes the execution dependency[1] in the bushy tree intractable, which in turn causes the problem of processor allocation much more complicated. To remedy this, we first devise in this paper a scheme to transform a bushy execution tree to an allocation tree in which each node denotes a pipeline. Then, using the concept of synchronous execution time [5], processors are allocated to the nodes in the allocation tree in such a way that inner relations in a pipeline can be made available approximately the same time, thus solving the execution dependency for the parallel execution of pipelined hash joins. Note that the method devised in this paper is to cope with inter-pipeline processor allocation so as to execute multiple pipelines in parallel, and should not be confused with the processor allocation *within* a pipeline studied in [15].

In addition, the approach of hash filtering is investigated to improve the query execution time. A study on using hash filters to improve the execution of sort-merge joins can be found in [3]. Note that depending on the cost and the benefit of hash filters, there are various schemes to determine the hash filter generation. Performance studies via a detailed simulation are conducted to demonstrate the importance of processor allocation and to evaluate different schemes using hash filters. Simulation results show that processor allocation based on the allocation tree significantly outperforms that based on the original bushy tree, showing the advantage of performing the proposed tree transformation to deal with processor allocation for pipelined hash joins. Among all schemes for hash filtering evaluated, the one to build hash filters only for inner relations emerges as a winner. It is experimentally shown that processor allocation is in general the dominant factor to performance, and the effect of hash filtering becomes more prominent as the number of relations in a query increases.

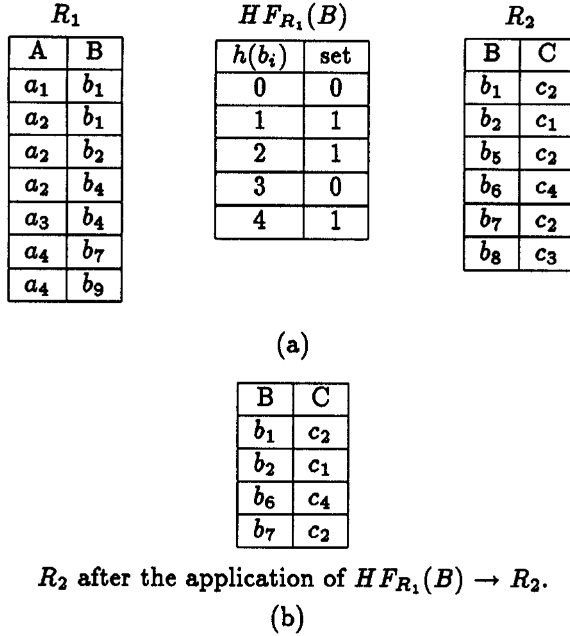
The rest of this paper is organized as follows. Preliminaries are given in Section 2. Schemes for processor allocation and hash filtering are presented in Section 3. Section 4 describes the underlying system and cost model. Performance studies on various schemes are conducted in Section 5 via simulation. This paper concludes with Section 6.

2 Preliminaries

We use $|R_i|$ to denote the cardinality of a relation R_i and $|A|$ to denote the cardinality of the domain of an attribute A. A query is assumed to have the form of conjunctions of equi-join predicates. We focus on the execution of complex queries [25], i.e., queries involving many relations. Notice that such complex queries can become frequent in real applications due to the use of views. The architecture assumed in the performance study in Section 4 is a multiprocessor system with distributed memories and shared disks containing database data. Barring any tuple placement skew, the scheme developed in this paper is applicable to the shared-nothing architecture where each disk is accessible only by a single node. A pipeline of hash joins is composed of several stages, each of which is associated with one join operation. The relation in a hash join that is loaded into memory to build the hash table is called the *inner relation*, while the other relation, whose tuples are used to probe the hash table, is called the *outer relation*. A detailed description and the advantage of pipelined hash joins can be found in [4].

Both CPU and I/O costs of executing a query are considered in our study. CPU cost is determined by the pathlength, i.e., the total number of tuples processed multiplied by the number of CPU instructions required for processing each tuple. A parameter on CPU speed (i.e., MIPS) is used to compute the CPU processing time from the number of CPU instructions incurred. I/O cost for processing a query is determined by disk service time per page multiplied by the total number of page I/O's. As such, we can appropriately vary the CPU speed to take into consideration both CPU bound and I/O bound query processing, and study the impact of processor allocation and hash filtering in both cases. A detailed description on the cost of hash joins and system parameters used is given in Section 4. In addition,

[1] Execution dependency means that a pipeline cannot be performed until all of its input relations are available.



R_1

A	B
a_1	b_1
a_2	b_1
a_2	b_2
a_2	b_4
a_3	b_4
a_4	b_7
a_4	b_9

$HF_{R_1}(B)$

$h(b_i)$	set
0	0
1	1
2	1
3	0
4	1

R_2

B	C
b_1	c_2
b_2	c_1
b_5	c_2
b_6	c_4
b_7	c_2
b_8	c_3

(a)

B	C
b_1	c_2
b_2	c_1
b_6	c_4
b_7	c_2

R_2 after the application of $HF_{R_1}(B) \rightarrow R_2$.

(b)

Figure 1: An example of the use of hash filters.

we assume for simplicity that the values of attributes are uniformly distributed over all tuples in a relation and that the values of one attribute are independent of those in another. Thus, the cardinalities of resulting relations of joins can be estimated according to the formula used in prior work [5]. In the presence of data skew, we only have to modify the corresponding formula accordingly [10], which, however, will not affect the relative performance of the schemes evaluated.

A hash filter (HF) built by relation R_i on its attribute A, denoted by $HF_{R_i}(A)$, is an array of bits which are initialized to 0's. Let $R_i(A)$ be the set of distinct values of attribute A in R_i, and h be the corresponding hash function employed. The k-th bit of $HF_{R_i}(A)$ is set to one if there exists an $a \in R_i(A)$ such that $h(a) = k$. Similar to the effect of semijoins, it can be seen that before joining R_i and R_j on their common attribute A, probing the tuples of R_j against $HF_{R_i}(A)$ and removing non-matching tuples will reduce the number of tuples of R_j to participate in the join. The join cost is thus reduced. An illustrative example of the use of hash filters can be found in Figure 1, where an $HF_{R_1}(B)$ is built by R_1 and applied to R_2, with the corresponding hash function $h(b_i)= i \bmod 5$. It can be verified that after the application of $HF_{R_1}(B)$, R_2 is reduced to the one given in Figure 1b, thus reducing the join cost of $R_1 \bowtie R_2$. In this paper, we shall develop an efficient scheme to interleave a bushy execution tree with hash filters to minimize the query execution cost for hash joins. Let $HF_{R_i}(A) \rightarrow R_j$ denote the application of a hash filter generated by R_i on attributed A to R_j. Note that the reduction of R_j by $HF_{R_i}(A) \rightarrow R_j$ is proportional to the reduction of $R_j(A)$. The estimation on the size of the relation reduced is thus similar to estimating the reduction of the projection on the corresponding attribute. Let $\rho_{i,A}$ be the reduction ratio by the application of $HF_{R_i}(A)$, and the cardinality of R_j after $HF_{R_i}(A) \rightarrow R_j$ can be estimated as $\rho_{i,A}|R_j|$. Clearly, the determination of $\rho_{i,A}$ depends on the size of a hash filter because, as shown in Figure 1, different attribute values may be hashed into the same hash entry. As pointed out in [3], hashing $k = |R_i(A)|$ different values into a hash filter of m bits is similar to the experiment of drawing k balls from m different balls with replacement. The following proposition, which is used to determine the effect of hash filters in our simulation, thus follows.

Proposition 1: The reduction ratio by the application of $HF_{R_i}(A)$, $\rho_{i,A}$, can be formulated as

$$\rho_{i,A} = \begin{cases} 1 - (1 - \frac{1}{m})^{|R_i(A)|}, & \text{for } m < |A|, \\ \frac{|R_i(A)|}{|A|}, & \text{for } m \geq |A|, \end{cases} \quad (1)$$

where $R_i(A)$ is the set of distinct values of attribute A in R_i, and m is the number of hash entries in a hash filter.

In addition, following the analysis in [3], we assume that the number of distinct values of a non-filtered attribute remains the same after a hash filter application in our study.

3 Algorithmic Aspects

In this section, we shall first derive a processor allocation scheme and then introduce methods to generate hash filters for a bushy execution tree.

3.1 Deriving and Utilizing Allocation Trees

As mentioned earlier, to exploit the opportunity of pipelining for hash join execution, one has to identify and execute a sequence of hash joins in the bushy tree in a way of pipelining. However, such regrouping of joins makes the execution dependency in the bushy tree intractable. Consequently, we shall first transform a bushy execution tree to an allocation tree where each node denotes a pipeline. Then, using the concept of synchronous execution time, processors are allocated to the nodes in the allocation tree in such a way that inner relations in a pipeline can be made available approximately the same time. Idleness of processors can thus be minimized.

To transform a bushy tree to an allocation tree, we first identify the groups of joins in the bushy tree

that could be pipelined. Then, an allocation tree can be obtained from the original bushy tree by merging each group of joins together. For example, suppose we determine six groups of joins to be pipelined as shown in Figure 2. By merging each pipeline in Figure 2 into a single node, we obtain the corresponding allocation tree as given in Figure 3. Next, we determine the number of processors allocated to each node (pipeline) in the allocation tree in the manner of top down. Clearly, all processors are allocated to the pipeline associated with the root in the allocation tree, say S_1 in Figure 3, since it is the last pipeline to be performed. Those processors allocated to the pipeline on the root are then partitioned into several clusters which are assigned to execute the pipelines associated with the child nodes of the root in the allocation tree in such a way that those pipelines can be completed approximately the same time. In other words, processors are so allocated that input relations for the root pipeline, say S_1 in Figure 3, can be available about the same time to facilitate the execution of S_1. The above step for partitioning the processors for the root is then applied to all internal nodes in the allocation tree in a top down manner until each pipeline is assigned with a number of processors. Specifically, define the *cumulative execution costs* of a node in the allocation tree as the sum of the execution costs of all pipelines in the subtree under that internal node. Let S_i be a pipeline associated with a node in the allocation tree T_A, and $C(S_i)$ be the set of child nodes of S_i in T_A. Denote the cost of executing S_i as $W(S_i)$. Then, the cumulative execution cost of the node with S_i, denoted by $CW(S_i)$, is determined by,

$$CW(S_i) = W(S_i) + \sum_{S_j \in C(S_i)} CW(S_j). \tag{2}$$

Note that the cumulative execution cost of each pipeline can be determined when the original bushy tree is built bottom up. Then, it is important to see that to achieve the synchronous execution time, when partitioning the processors of a node into clusters for its child nodes, one has to take into account the cumulative execution costs of the child nodes, rather than their individual execution costs. Denote the set of processors allocated to perform the pipeline S_x as $P(S_x)$, and use $\#P(S_x)$ to represent the number of processors in $P(S_x)$. We have,

$$\#P(S_x) = \lceil \#P(S_i) \frac{CW(S_x)}{\sum_{S_j \in C(S_i)} CW(S_j)} \rceil. \tag{3}$$

Formally, the processor allocation scheme based on an allocation tree can be described below.

Algorithm *G*: Allocating processors to a bushy tree utilizing pipelined hash joins.

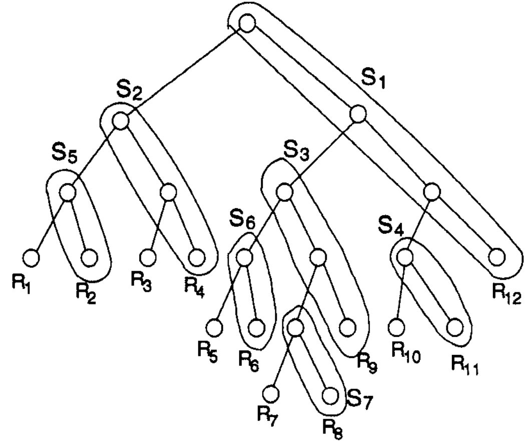


Figure 2: Identifying pipelines in a bushy tree.

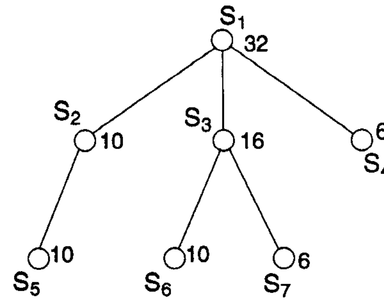


Figure 3: Illustration of an allocation tree.

Step 1: A join sequence heuristic is applied to determine a bushy execution tree T.

Step 2: From the given bushy tree T, determine the corresponding allocation tree T_A by merging relations in each pipeline together.

Step 3: Based on Eq.(2), determine the cumulative workload of each node in T_A in a bottom up manner.

Step 4: Using Eq.(3), allocate processors to each node in T_A in a top down manner.

For an illustrative purpose, given a total of 32 processors, a possible processor allocation for the allocation tree in Figure 3 is shown by a set of numbers labeled next to nodes in the tree. As mentioned earlier, these numbers are so determined that all child nodes (pipelines) of each internal node can be completed approximately the same time. It is worth mentioning that when the number of processors passed to an internal node in a lower level of the tree is too few

to be further partitioned for efficient execution of pipelines, sequential execution for the pipelines in its child nodes could be employed for better performance. Note that the method described above determine the inter-pipeline processor allocation to execute pipelined hash joins. Clearly, how to distribute processors across many stages within a pipeline is also an important issue, whose discussion is beyond the scope of this paper. Readers interested in processor allocation within a pipeline are referred to [15]. As it will be seen in Section 5 later, processor allocation based on the allocation tree outperforms that based on the original bushy tree. This result indicates the very difference between the pipelined hash join and the sort-merge join as far as processor allocation is concerned, and justifies the necessity of performing the proposed tree transformation to deal with processor allocation for pipelined hash joins.

3.2 Interleaving a Bushy Tree with HF's

To further improve the parallel execution of hash joins, we would like to employ hash filters to reduce the cost of each individual join. Note that a hash filter has to be received in time for its processing, which is, however, difficult to achieve due to the nature of parallel execution of hash joins. It can be seen that based on the processor allocation in algorithm G we have $S_j \in C(S_i) \Rightarrow P(S_j) \subseteq P(S_i)$, meaning that a pipeline in a higher level of the allocation tree will be executed by those processors allocated to its offspring. This feature indeed resolves the timing constraint described above, since under such processor allocation, a hash filter received late can still be applied to a later pipeline which is executed by the same cluster of processors, thus avoiding incurring any transmission for hash filters among processors. There are many methods conceivable to generate hash filters. For example, one method is to generate and apply hash filters to the bushy tree first and then to proceed the normal execution. Such a scheme is termed "early generation" (denoted by EG) in what follows, whose algorithmic form can be described below.

Algorithm EG: Interleaving a bushy tree T with HF's.

HF Sender: **for** each leaf node R_i in T
/* Let R_i be a relation in pipeline S_j. */
begin
  Let J_{att} be the set of all join attributes in R_i.
  if ($J_{att} \neq \phi$)
  begin
    Scan R_i, and $\forall A \in J_{att}$, build $HF_{R_i}(A)$ by $P(S_j)$.
    Send $HF_{R_i}(A)$ to $P(S_k)$, where S_k contains a relation joining with R_i on A.
  end
end
HF Receiver: **for** each leaf node R_i in T
begin
  if (R_i receives all HF's for its join attributes) **then**
    R_i applies HF's to filter out non-matching tuples.
    if (R_i is an inner relation) **then**
    building the hash table
    else starting tuple probing when inner relations are available.
end

The first conditional statement in HF Sender to set up J_{att} assures that only necessary hash filters will be generated and applied to other relations. Also, it can be seen that a relation will be scanned at most once to build HF's for attributes in J_{att}. Every relation, after receiving and utilizing all its filters, starts its normal operations.

Depending on the cost and the benefit of hash filters, there are many schemes to determine the hash filter generation. To provide more insights into the approach of hash filtering, extensive simulation will be conducted in Section 5 to evaluate various schemes using hash filters. Owing to the nature of hash joins, instead of being generated in advance, hash filters can be built together with the hash join operations to reduce the overhead associated. Specifically, hash filters from inner relations are built in their table-building phases and those from outer relations are built in their tuple-probing phases. Such an approach will be referred to as scheme CG, where CG stands for "complete generation." Also, as will be evaluated in Section 5, hash filters can be generated from inner relations only to reduce the cost of hash filter generation while attaining the desired reduction effect. This alternative is denoted by IG, standing for "inner relation generation." The conventional approach without using hash filters, denoted by NF (i.e., "no hash filters"), will also be implemented for comparison purposes.

4 System and Cost Model

In this section, we describe the underlying system and cost model, based on which the experiments will be conducted to study the relative performance of various processor allocation and hash filtering schemes.

4.1 Overview

The architecture assumed in this study is a multiprocessor system with distributed memories and shared disks containing database data. It is also assumed that a processor activates one I/O process for every relation scan to read its portion of the relation from disk. Our

goal is to evaluate the performance of processor allocation and hash filtering schemes in a variety of complex query environments. The performance model consists of three major components: *Request Generator*, *Compiler*, and *Executor*. Request Generator is responsible for generating query requests as follows. The number of relations in a query is determined by an input parameter, sn. Relation cardinalities and join attribute cardinalities are determined by a set of parameters: R_{card}, $carv$, $f_d(R)$, A_{card}, $attv$, and $f_d(A)$. Relation cardinalities in a query are computed from a distribution function, $f_d(R)$, with a mean, R_{card}, and a deviation, $carv$. Cardinalities of join attributes are determined similarly by A_{card}, $attv$, and $f_d(A)$. There is a predetermined probability, p, that an edge (i.e., a join operation) exists between any two relations in a given query graph. The larger p is, the larger the number of joins in a query will be. Note that some queries so generated may have disconnected query graphs. Without loss of generality, only queries with connected query graphs were used in our study, and those with disconnected graphs were discarded. Compiler takes a query request from Request Generator and produces a query plan in the form of a bushy tree. The bushy plan tree is determined by the *minimum cost* heuristic described in [5] that tries to perform the join with the minimal cost first.

Executor traverses the query plan tree and carries out join operations in parallel according to join sequence determined by the Compiler. Depending upon the hash filtering schemes simulated, hash filters of join attributes are generated at different stages of query execution. While always generated from base relations, hash filters can be applied to both base relations and intermediate relations. After being generated, a hash filter is sent to all nodes (processors) that will perform join operation on the pipeline containing the corresponding relation. For every hash filter received, it is applied to the corresponding base relation if the filter is received before the start of the relation scan. Otherwise, a filter received will usually be applied to the intermediate relation generated at the end of a pipelined join operation. However, in the worst case, a hash filter may not be received in time for applying to either base or intermediate relation, in which case, the hash filter will be discarded. The dynamic nature of a bushy tree execution is thus captured in our simulation.

In the pipelined hash join operation, a tuple from outer relation can successively probe hash tables of multiple inner join relations in the same pipeline, and a pipelined join operation will not start until all inner relations of the pipeline have completed building hash tables in memory. To minimize processor idle time and maximize the performance benefit of parallel query execution, processors should be allocated to joining relations in such a way that the table-building phase of all inner relations of a pipeline are completed approximately at the same time. To achieve this, the number of processors allocated to a join node is determined in a *top down* manner and the processors assigned to the parent node are divided among the child nodes according to the cumulative execution costs of the child nodes. In our simulation model, processor allocation is done as part of query compilation. Therefore, a query plan node in our experiment contains the number of processors allocated to execute that node, in addition to other information on the database profile. Note that in practice, allocation of processors to a join node could be deferred to runtime to provide more flexibility.

Two schemes for assigning processors to join nodes are comparatively studied in this paper. In the first scheme, the cumulative execution cost is computed for every node of the query plan tree generated by the Compiler. Processors are then allocated to execute a query plan node according to the cumulative execution cost of the node. In this scheme, the cumulative execution cost of a node is computed as the sum of the execution cost of joining the relations associated with its two child nodes plus the cumulative execution costs of its two child nodes as described in Section 3. The cumulative execution cost of a leaf node is defined to be the cost of scanning the base relation. Henceforth, we shall refer to this processor allocation scheme as **BOT** (standing for Based on the Original Tree) scheme. In the second scheme, by grouping (join) nodes within each pipeline together, a query plan tree is first transformed to an allocation tree. Cumulative workload is computed for each node of the allocation tree, and processors are then allocated to nodes based on the allocation tree. This processor allocation scheme is henceforth referred to as **BAT** (standing for Based on the Allocation Tree) scheme.

4.2 Cost Model

Our model computes both CPU and I/O costs of executing a query. For ease of presentation, we assume that each node (processor) has large enough physical memory to hold hash tables of all inner relations of a pipeline in memory at the same time (i.e., bucket overflow will not occur). The effect of bucket overflow is believed not to affect the relative performance of the processor allocation schemes we shall evaluate. Also, the intermediate relation generated by a completed pipelined join operation is assumed to be written to disks. When an intermediate relation is to be used by a later (pipelined) join operation, it is read from the disks and, at the same

time, the hash table corresponding to the next join attribute is built in memory.

The number of CPU instructions executed to read a data page from a disk is denoted by I_{read}, and that to write a data page to a disk is denoted by I_{write}. The cost of extracting a tuple from a page in memory is denoted by I_{tuple}. The cost of building a hash table is determined by multiplying the total number of tuples processed by the number of CPU instructions each tuple needs for table-building (i.e., I_{build}). Similarly, the cost of probing a hash table is determined by multiplying the total number of tuples processed by the number of CPU instructions each tuple needs for tuple-probing (i.e., I_{probe}). Thus, the total CPU cost of building a hash table in memory for a relation of N tuples, including the cost of reading the relation from disks and that of extracting tuples from pages, is equal to $I_{read} \times N/p_{size} + I_{tuple} \times N + I_{build} \times N$, where p_{size} is the number of tuples per page. Also, the CPU cost of carrying out the tuple-probing phase of a join operation, with the outer relation size of N_p tuples, is equal to $I_{probe} \times N_p$, which is independent of the inner relation size N. Consequently, to execute a pipeline of n joins, the total CPU cost is then equal to $\sum_{i=0}^{n} N_i/p_{size} \times I_{read} + \sum_{i=0}^{n} N_i \times I_{tuple} + \sum_{i=1}^{n} N_i \times I_{build} + \sum_{i=1}^{n} N_{p_i} \times I_{probe}$, where N_i is the inner relation size of i-th join (except N_0 that is the size of the outer relation of the first join operation) and N_{p_i} denotes the outer/probing relation size of the i-th join operation in the pipeline. When $i = 1$, N_{p_i} is equal to N_0. For $i > 1$, N_{p_i} is the size of the result relation generated by the $(i-1)$-th join of the pipeline. Note that the above cost does not include the spooling of intermediate relation to disks at the end of a pipelined join. The cost of spooling is equal to $N_{p_{n+1}}/p_{size} \times I_{write}$ instructions, which are added to the CPU cost listed above for all pipelined join operations except when the pipeline produces the final result for the query.

The CPU processing time for executing a query is obtained from dividing the total number of CPU instructions per query by the CPU speed, P_{speed}. By separating the pathlength per query and the CPU speed, we have the flexibility of varying the CPU speed to make a query execution either CPU bound or I/O bound, and studying the impact of using hash filters in both cases. I/O cost for reading (respectively, writing) a relation of N tuples is determined by disk service time per page, t_{pio}, multiplied by the total number of page read (respectively, written). To execute a pipeline of n joins, $n+1$ relations are read from disks and the total I/O cost for reading relations is thus equal to $\sum_{i=0}^{n} N_i/p_{size} \times t_{pio}$. The intermediate relation generated by a pipelined join is always spooled to disks except the pipeline produces the final result. This increases the I/O cost by an amount of $N_{p_{n+1}}/p_{size} \times t_{pio}$. As before, $N_{p_{n+1}}$ denotes the size of the result relation generated by the n-th (last) join of the pipeline.

For schemes that generate and apply hash filters, the CPU cost of generating a hash filter from a relation of size N is computed by multiplying I_{hash} by N while the cost of applying a hash filter to a relation of size N is computed by multiplying I_{apply} by N. I_{hash} is the number of CPU instructions required to generate a hash value from an input tuple and set the corresponding bit in the hash filter, and I_{apply} is the number of instructions needed to check whether an attribute value of a tuple has a match in the filter, and if that bit is set, add the tuple to a temporary relation to be joined later. Note that hash filter generation phase can be combined with the base relation scan for join operation, thus avoiding I/O overhead in hash filter generation (the IG and CG schemes). On the other hand, if all hash filters are generated in a separated phase, prior to the start of the first join operation of the query (the EG scheme), N/p_{size} additional I/O's per relation are required. It is worth mentioning that, in our simulation model, hash filters are implemented as bit-vectors and can in general fit in memory, thus minimizing extra I/O's required for maintaining them.

4.3 Parameter Setting

We select queries of five sizes, i.e., queries with 4, 8, 12, 16, and 20 relations. This set of selections covers a wide spectrum of query sizes ranging from a simple three way join to a more than twenty way join. For each query size, 500 query graphs were generated, and as mentioned in Section 4.1, only queries with connected query graphs are used in our study.

To conduct the simulation, [1], [7], [12], and [20] were referenced to determine the values of simulation parameters. Table 1 summarizes the parameter settings used in the simulation. The number of processors in the system is set to 32 and the speed of the processor is assumed to be 3 MIPS. The number of CPU instructions for either reading or writing a page is set to 5000, and that for extracting a tuple from page in memory is set to 300. Applying hash function to an attribute to build either hash table or hash filter is assumed to take 100 instructions each while probing hash table or probing hash filter to filter out non-matching tuples is assumed to consume 200 instructions each. Each page is assumed to contain 40 tuples and disk service time per page (both read and write) is assumed to be 20 milliseconds. The size of a relation varies from 700K to 1300K tuples while each attribute has from 600K to 1000K distinct

parameters	setting
n_p	32
P_{speed}	3 MIPS
I_{read}	5000
I_{write}	5000
I_{tuple}	300
I_{build}	100
I_{prob}	200
I_{hash}	100
I_{apply}	200
p_{size}	40 tuples
t_{pio}	20 ms
R_{card}	1M tuples
A_{card}	800K
carv	± 100K - 300K
attv	± 100K - 200K
$f_d(A)$	uniform
$f_d(R)$	uniform

Table 1: Parameters used in simulation.

values. Finally, the distribution functions for relation cardinality and attribute cardinality are both assumed to be uniform.

5 Simulation Results

In the simulation program, which was coded in C, the action for each individual relation to go through join operation, with the corresponding hash filter generation and application, was simulated. For each query in the simulation, four hash filtering schemes, i.e., NF (no filter), EG (early generation), IG (inner generation) and CG (complete generation), were applied to execute the query. The CPU cost, I/O cost, and total response time for each scheme were obtained.

The experiment was carried out in two stages. In the first stage, we studied the relative performance of different processor allocation schemes. Specifically, we explored the benefit, in terms of reduction in query response time, of allocating processors to join nodes based on allocation tree (BAT) in a parallel database environment. In the second stage, we fixed the processor allocation scheme to BAT and ran a set of experiments to study the relative performance of the four different hash filtering schemes.

5.1 On Processor Allocation

Exp. 1: Processor allocation schemes

Performance of BOT and BAT processor allocation schemes is evaluated by this experiment, where both *attv* and *carv* were set to 100K and the number of processors was set to 32. The number of processors allocated to execute a join node is determined by the processor allocation scheme employed. Figure 4 shows the average response times of the queries for the NF method using BOT and BAT allocation schemes. We deliberately turned off hash filter application in this experiment to demonstrate the importance of processor allocation in multi-processor database systems. In this figure, the ordinate is the response time in milliseconds and the abscissa denotes the number of relations in a query. As illustrated in Figure 4, the response time with BAT is significantly lower than that with BOT in all queries evaluated, clearly showing the advantage of performing the proposed tree transformation. Using BOT scheme, the average response time for a join query involving four relations (i.e., $sn = 4$) is about 354 seconds while it is 3874 seconds for the case of $sn = 20$. By employing tree transformation before processor allocation, the average response times for $sn = 4$ and $sn = 20$ by BAT were reduced to 147 seconds and 1235 seconds, respectively, showing an improvement from 50% to 70%.

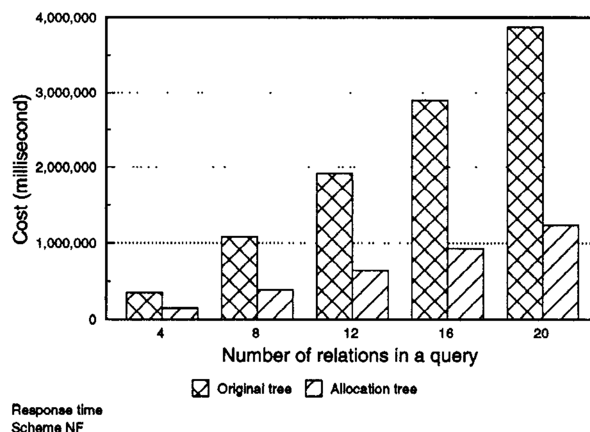


Figure 4: The response time for the NF scheme.

This experiment shows that the BOT processor allocation scheme, which was demonstrated to provide good performance with sort-merge join methods [5], does not perform well with pipelined hash join method. Note that with pipelined hash join, a join operation does not start until all hash tables of the pipeline are materialized in memory. It is therefore advantageous to allocate processors to join nodes in such a way that all inner relations are materialized around the same time. Clearly, this can be achieved by the tree transformation we devised in Section 3, which groups together all inner relations of a pipeline and considers them as a whole for

processor allocation. By taking this nature of pipelined hash joins into consideration, BAT leads to a significantly better performance than BOT scheme.

5.2 On HF Schemes

In the second stage of our simulation study, the processor allocation scheme was fixed to BAT due to its superiority to BOT. Our goal is to study the effectiveness of alternative HF schemes for queries with varying complexity. As described in Section 2, the EG scheme generates hash filters from all base relations before the start of actual join operation, and then applies hash filters to base relations during join operations. This scheme provides the maximum reduction effect of hash filtering because all hash filters are available (and can thus be applied) before the start of any join operation. However, EG also incurs the highest overhead because it needs to go through one extra round of disk I/O to read the base relations from disks in order to generate the hash filters. IG was designed to minimize the overhead associated with the hash filter generation and CG was designed to maximize the effect of hash filtering. With IG, hash filters are generated from inner base relations only and they are generated at the time when inner relations are read from disks to build hash tables in memory. Consequently, no extra I/O is incurred by IG and the overhead of applying hash filter is thus minimized. With CG, hash filters are also generated from all base relations. Unlike EG, CG generates hash filters from a relation when the relation is to be joined next, and thus no extra I/O is incurred. Note that because hash filters from outer relations are not generated by CG until the relations are to be joined next, these filters might not be received in time to be used by partner relations. As a result, though generating the same number of hash filters as EG, CG in general applies fewer hash filters than EG. However, as it will be shown later, despite its fewer applications of hash filters, CG will outperform EG due to its timely generation of hash filters with better filtering effect and also its saving in I/O cost.

Exp. 2: Low variance, carv=100K and attv=100K

In this experiment, relation cardinality ranges from 900K to 1100K tuples while attribute cardinality ranges from 700K to 900K. The average CPU cost, I/O cost, and response time for this experiment are shown in Figures 5, 6, and 7, respectively. Figures 5 and 6 show that with 3 MIPS CPU, these queries using the pipelined hash join method are I/O bound. The 20 ms page I/O time setting assumes no prefetching or disk buffering (e.g., reading one track at a time). The experiment also assumes that one I/O process is activated by each processor for every relation scan. Note that this experiment could become CPU bound if disk buffering or parallel I/O strategy (activate multiple I/O processes per processor for every relation scan) was used.

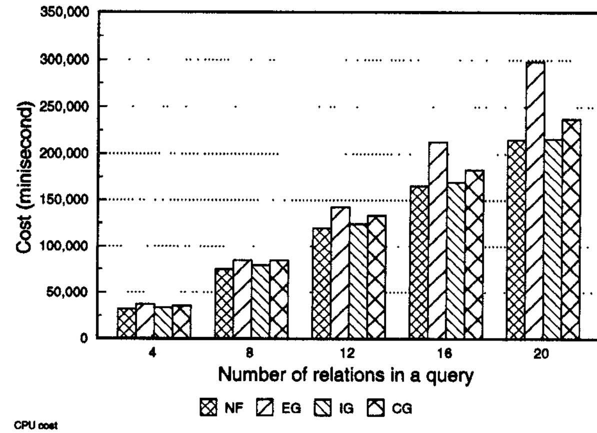


Figure 5: The CPU cost for each scheme based on AT.

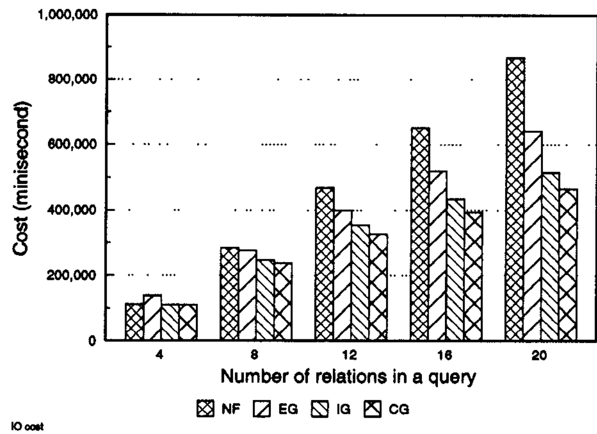


Figure 6: The IO cost for each scheme based on AT.

Figure 5 shows that applying hash filters results in a significantly larger CPU cost with the EG method. It can also be seen that the CPU cost with CG is slightly larger than that with NF. IG consumes approximately the same amount of CPU resource as NF. This experiment shows that, as far as the CPU cost is concerned, the benefit of applying hash filter (i.e., size reduction) is overshadowed by the cost of generating and applying hash filters with both EG and CG. Note that without bucket overflow, the CPU cost of a hash join is linearly proportional to the size of two input relations. In the simulation, we assume that intermediate relations are not spooled to disks except at the end of a pipelined join and that bucket overflow does not occur. Conse-

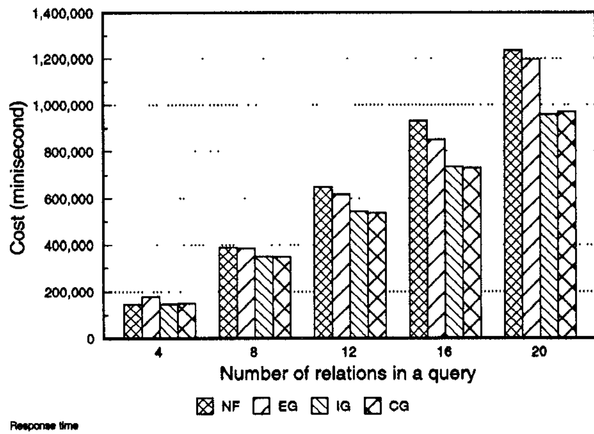


Figure 7: The average response time for each scheme based on AT.

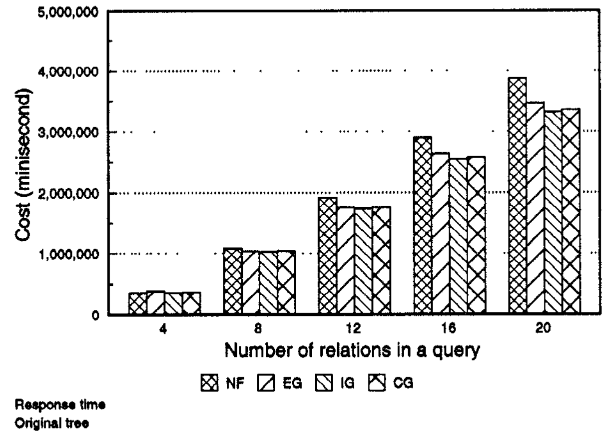


Figure 8: The average response time for each scheme based on PN.

quently, the benefit of reduction in intermediate relation size appears only at the end of a pipelined join, resulting in smaller benefit of applying hash filter. If more intermediate relations are spooled to disks, applying hash filter will be more favorable.

Figure 6 shows that applying hash filters results in a slight performance improvement in I/O cost when sn is small ($sn \leq 8$). The improvement increases significantly as the number of relations in a query increases. This is because the number of pipelines increases as sn increases and, as explained earlier, the benefit of size reduction thus becomes more prominent. It can also be seen from Figure 6 that CG performs the best among all schemes evaluated while NF is outperformed by all other schemes. As previously described, neither CG nor IG incurs any I/O overhead in generating and applying hash filters. Because more hash filters are generated and applied with CG, the size of an intermediate relation with CG, on the average, is smaller than that with IG. Consequently, CG provides the lowest I/O cost among all schemes evaluated while IG the second lowest. For EG, it applies more hash filters than all other schemes and thus achieves the maximum reduction effect. However, it scans the base relations one extra time to build the hash filters. As a result, the total I/O cost with EG is lower than that with NF, but larger than those with IG and CG.

Figure 7 shows that, except in the case of $sn = 4$, the total query response time can be reduced by applying hash filters. When sn =4, performance of IG and CG is similar to that of NF. With EG, however, the response time increases by about 21% compared to that with NF. When $sn = 20$, the response time is reduced, related to NF, by more than 20% with either IG or CG while the improvement is about 3% with EG. The result clearly demonstrates that both IG and CG are effective methods to reduce the response time of complex query execution using the pipelined hash join method, especially when sn is large. The experiment also shows that the extra system resource consumed by EG outweighs the benefit of size reduction when $sn \leq 8$. When sn is greater than 8, EG shows performance improvement over NF, but the reduction in response time is limited to be less than 5%.

For reference, Figure 8 shows the response time of the queries for HF schemes using the BOT processor allocation scheme. As indicated in the figure, IG remains to be the best scheme evaluated and the improvement over NF in response time is about 15% when $sn = 20$. Compared to Figure 7, Figure 8 shows that the benefit of applying hash filter is smaller with BOT than with BAT. This is because, with BOT, processor idle time (waiting for peers to complete preceding join operations) contributes to a significant portion of the total response time and, as a result, the saving in join execution time by applying hash filter becomes less significant.

Exp. 3: High variance, carv=300K and attv=200K

In this experiment, we increased the variance of relation cardinality from 100K to 300K and the variance of attribute cardinality from 100K to 200K. By changing the variances of relation and attribute cardinalities, the effectiveness of hash filters on hash join operations with varied cardinalities can be studied. Figures 9, 10, and 11 show, respectively, the CPU cost, the I/O cost, and the response time for each scheme. Similar to Figure

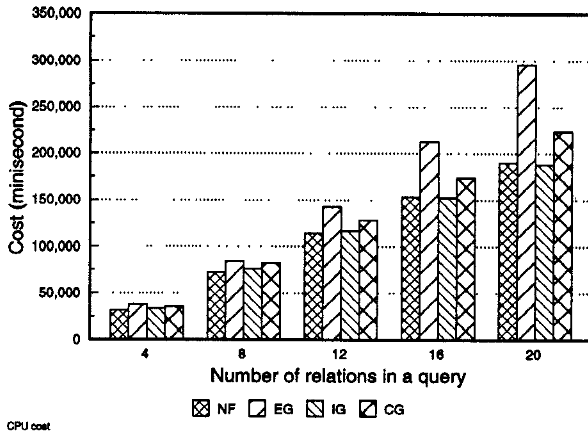


Figure 9: The CPU cost for each scheme, high variance.

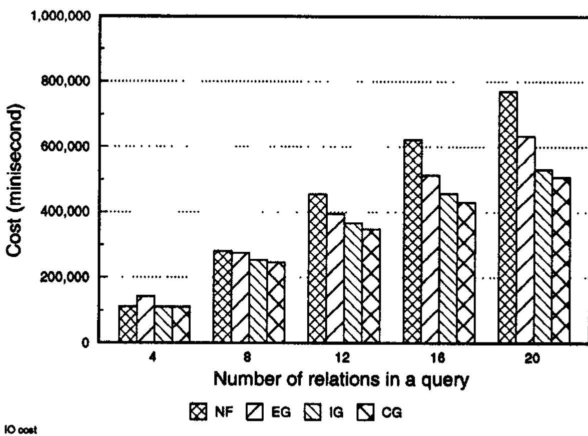


Figure 10: The IO cost for each scheme, high variance.

5, Figure 9 shows that EG consumes the most CPU resource while NF and IG the least. On the other hand, Figure 10 shows that, except when $sn = 4$, NF consumes the most disk resource while CG the least. Overall, as indicated in Figure 11, when $sn > 4$, both IG and CG generate noticeable performance improvement over NF. When $sn = 20$, applying hash filters can improve the response time by more than 20% with the IG scheme. Compared to the results in Exp. 2, these three figures indicate that the effectiveness of applying hash filter is stable when the variances of relation cardinalities and attribute cardinalities increase. As before, this experiment shows that IG is the best scheme among all schemes evaluated while CG is the second.

6 Conclusions

In this paper we explored two important issues, processor allocation and the use of hash filters, to improve

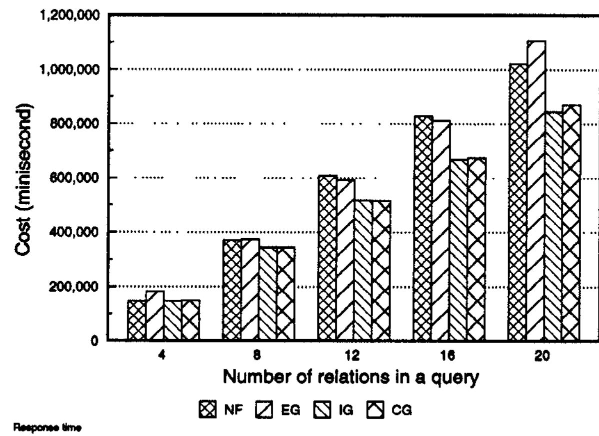


Figure 11: The average response time for each scheme, high variance.

the parallel execution of hash joins. Performance studies have been conducted to demonstrate the importance of processor allocation and to evaluate various schemes using hash filters via simulation. Simulation results showed that processor allocation based on the allocation tree significantly outperformed that based on the original bushy tree. Among all schemes for hash filtering evaluated, the one to build hash filters only for inner relations emerged as a winner. It was experimentally shown that processor allocation is in general the dominant factor to performance, and the effect of hash filtering becomes more prominent as the number of relations in a query increases.

References

[1] D. Bitton and J. Gray. Disk Shadowing. *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 331–338, September 1988.

[2] H. Boral, W. Alexander, et al. Prototyping Bubba, A Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, March 1990.

[3] M.-S. Chen, H.-I. Hsiao, and P. S. Yu. Applying Hash Filters to Improving the Execution of Bushy Trees. *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 505–516, August 1993.

[4] M.-S. Chen, M.-L. Lo, P. S. Yu, and H. C. Young. Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins. *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 15–26, August 1992.

[5] M.-S. Chen, P. S. Yu, and K.-L. Wu. Scheduling and Processor Allocation for Parallel Execution

of Multi-Join Queries. *Proceedings of the 8th International Conference on Data Engineering*, pages 58–67, February 1992.

[6] D. J. DeWitt and R. Gerber. Multiprocessor Hash-Based Join Algorithms. *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 151–162, August 1985.

[7] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

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

[9] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query Optimization for Parallel Execution. *Proceedings of ACM SIGMOD*, pages 9–18, June, 1992.

[10] D. Gardy and C. Puech. On the Effect of Join Operations on Relation Sizes. *ACM Transactions on Database Systems*, 14(4):574–603, December 1989.

[11] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. *Proceedings of the 1st Conference on Parallel and Distributed Information Systems*, pages 218–225, December 1991.

[12] H.-I. Hsiao and D. DeWitt. A Performance Study of Three High Availability Data Replication Strategies. *Proceedings of the 1st Conference on Parallel and Distributed Information Systems*, pages 79–84, December 1991.

[13] K. A. Hua, Y.-L. Lo, and H. C. Young. Including the Load Balancing Issue in the Optimization of Multi-Way Join Queries for Shared-Nothing Database Computers. *Proceedings of the 2nd Conference on Parallel and Distributed Information Systems*, pages 74–83, January 1993.

[14] Y. E. Ioannidis and Y. C. Kang. Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and its Implication for Query Optimization. *Proceedings of ACM SIGMOD*, pages 168–177, May 1991.

[15] M.-L. Lo, M.-S. Chen, C. V. Ravishankar, and P. S. Yu. On Optimal Processor Allocation to Support Pipelined Hash Joins. *Proceedings of ACM SIGMOD*, pages 69–78, May 1993.

[16] H. Lu, M.-C. Shan, and K.-L. Tan. Optimization of Multi-Way Join Queries for Parallel Execution. *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 549–560, September 1991.

[17] H. Lu, K. L. Tan, and M.-C. Shan. Hash-Based Join Algorithms for Multiprocessor Computers with Shared Memory. *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 198–209, August 1990.

[18] N. Roussopoulos and H. Kang. A Pipeline N-Way Join Algorithm Based on the 2-Way Semijoin Program. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):461–473, December 1991.

[19] D. Schneider. Complex Query Processing in Multiprocessor Database Machines. Technical Report Tech. Rep. 965, Computer Science Department, University of Wisconsin-Madison, September 1990.

[20] D. Schneider and D. J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. *Proceedings of ACM SIGMOD*, pages 110–121, 1989.

[21] D. Schneider and D. J. DeWitt. Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 469–480, August 1990.

[22] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. *Proceedings of ACM SIGMOD*, pages 23–34, 1979.

[23] J. Srivastava and G. Elsesser. Optimizing Multi-Join Queries in Parallel Relational Databases. *Proceedings of the 2nd Conference on Parallel and Distributed Information Systems*, pages 84–92, January 1993.

[24] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout. The Design of XPRS. *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 318–330, 1988.

[25] A. Swami. Optimization of Large Join Queries: Combining Heuristics with Combinatorial Techniques. *Proceedings of ACM SIGMOD*, pages 367–376, 1989.

[26] A. Wilschut and P. Apers. Dataflow Query Execution in Parallel Main-Memory Environment. *Proceedings of the 1st Conference on Parallel and Distributed Information Systems*, pages 68–77, December 1991.

[27] J. L. Wolf, J. T. Turek, M.-S. Chen, and P. S. Yu. Scheduling Multiple Queries on a Parallel Machine. *Proceedings of the ACM Sigmetrics Conference*, May 1994.

[28] M. Ziane, M. Zait, and P. Borla-Salamet. Parallel Query Processing in DBS3. *Proceedings of the 2nd Conference on Parallel and Distributed Information Systems*, pages 93–102, January 1993.