APPLYING HASH FILTERS TO IMPROVING THE EXECUTION OF BUSHY TREES

Ming-Syan Chen, Hui-I Hsiao and Philip S. Yu

IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract

In this paper, we explore an approach of interleaving a bushy execution tree with hash filters to improve the execution of multi-join queries. The effect of hash filters is evaluated first. Then, an efficient scheme to determine an effective sequence of hash filters for a bushy execution tree is developed, where hash filters are built and applied based on the join sequence specified in the bushy tree so that not only is the reduction effect optimized but also the cost associated is minimized. Various schemes using hash filters are implemented and evaluated via simulation. It is experimentally shown that the application of hash filters is in general a very powerful means to improve the execution of multi-join queries, and the improvement becomes more prominent as the number of relations in a query increases.

1 Introduction

In relational database systems, joins are the most expensive operations to execute. Applications involving decision support and complex objects usually have to specify their desired results in terms of complex multi-join queries, and some queries may take hours to complete, thus degrading the system performance. Apparently, such problems could be aggravated by the increases in database size and query complexity nowadays [18] [28]. As a result, it has become imperative to develop solutions for efficient execution of multi-join queries for future database management [3] [8] [16] [19].

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 left-deep and right-deep trees 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 sparked off many subsequent studies. Several schemes have been proposed to develop sequential join sequences. A heuristic scheme to optimize multi-join queries with an enlarged search space was proposed in [14]. The benefit of using such techniques as simulated annealing and iterative improvement to tackle large search space for query optimization was studied in [23] and [24]. In [11], a collection of good sequential plans was first obtained based on the buffer space, and parallelization of this collection of plans was then explored.

The bushy tree join sequences, on the other hand, did not attract as much attention as sequential ones in the last decade since it was generally deemed sufficient, by many researchers, to explore only sequential join sequences for desired performance. This can be in part explained by the reasons that in the past the power/size of a multiprocessor system was limited, and that the query structure used to be too simple to require further parallelizing as a bushy tree. It is noted, however, that these two limiting factors have been phased out by the rapid increase in the capacity of multiprocessors and the trend for queries to become more complicated nowadays [28], thereby justifying the necessity of exploiting bushy trees. Consequently, 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 [13] to shed some light on the complexity of choosing left-deep and bushy trees. An

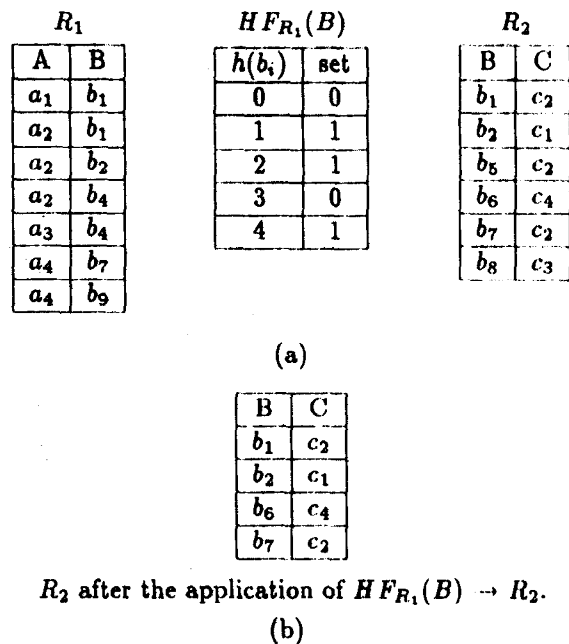
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 19th VLDB Conference, Dublin, Ireland, 1993

integrated approach dealing with both intra-operator and inter-operator parallelism was presented in [17], where a greedy scheme taking various join methods and their corresponding costs into consideration was proposed. As an extension to [11], an algorithm handling processor scheduling in a bushy tree was proposed in [10], where the inter-operator parallelism is achieved by properly selecting IO-bound and CPU-bound task mix to be executed concurrently. For efficient solutions, only schemes that execute at most two tasks at a time were explored in [10]. A two-step approach to deal with join sequence scheduling and processor allocation for parallel query processing using sort-merge joins was devised in [6]. Pipelining hash joins in a bushy tree and processor allocation within each pipeline were studied in [4] and [15], respectively. In addition, various query plans in processing multi-join queries in a shared-nothing architecture were investigated in [20].

While most prior work on inter-operator parallelism focused on the execution tree generation to minimize the query execution cost, there is relatively little result reported on exploiting the structure of a query tree to further reduce each individual join cost. It has been shown that the cost of executing a join operation can mainly be expressed in terms of the cardinalities of relations involved. In view of this, one would naturally like to remove unnecessary tuples and reduce the cardinalities of relations involved before a join to minimize the join cost. As semijoin has traditionally been relied upon to reduce the amount of inter-site data transmission required for distributed query processing [2] [5], the technique of hash filtering can be applied in a parallel database environment to reduce the relation cardinalities. Note, however, that prior work on hash filters (or called bit-vector filters) only considered their use on the joining attribute due mainly to the focus on linear execution trees [1] [7] [25][1], thus not fully taking advantage of the opportunity for utilizing multiple hash filters to reduce a single relation. As can be seen later, such an opportunity is made available by the execution of a bushy tree, and can lead to a very significant reduction effect on relation cardinalities, thereby greatly improving the execution of multi-join queries. Consequently, we explore in this paper the approach of interleaving a bushy execution tree with hash filters (HF's) to minimize the query execution time. It is worth mentioning that the algorithm we propose aims at improving the execution of a bushy tree, thus providing a solution to an increasingly important problem.

[1] Note that in dealing with a linear execution tree, one usually has only two joining relations residing in memory at a time, thus limiting the applicability of hash filters to the joining attribute.



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.

Due to the complexity of a bushy tree, hash filters built and applied in different execution stages can have very different costs and reduction effects. How to timely and appropriately build hash filters so as to minimize their cost as well as optimize their effect is a very important issue, and hence taken as the objective of this study.

A hash filter 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 $R_1 \bowtie R_2$. Note that the effect of hash filters is more complicated than that of semijoins, since hash collision can occur for different attribute values (such as b_1 and b_6 in Figure 1a) when a hash filter is built. In this paper, we shall evaluate the effect of hash filters first, and then develop an efficient scheme to interleave a bushy

execution tree with hash filters to minimize the query execution cost. As mentioned earlier, hash filters built in different execution stages of a bushy tree can have different costs and reduction effects. In view of this, the proposed scheme will assign a join sequence number to each join operation in the bushy tree when the tree is being built at the compile time[2]. The join sequence numbers specify the order of the joins to be carried out. Then, based on the join sequence in the bushy tree, hash filters are built and applied, cost-effectively, so that not only is the reduction effect optimized but also the cost associated is minimized. Several illustrative examples will be given. Extensive performance studies are conducted to evaluate various schemes using hash filters via simulation. It is experimentally shown that the application of hash filters is in general a very powerful means to improve the execution of multi-join queries, and the improvement 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. The effect of hash filters and the proposed scheme are presented in Section 3. Performance studies on various schemes using hash filters are conducted in Section 4 via simulation. This paper concludes with Section 5.

2 Preliminaries

We assume that a query is of the form of conjunctions of equi-join predicates. A join query graph can be denoted by a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. Each vertex in a join query graph represents a relation. Two vertices are connected by an edge if there exists a join predicate on some attribute of the two corresponding relations. 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. As in most prior work on the execution of database operations, we assume that the execution time incurred is the primary cost measure for the processing of database operations. Also, we focus on the execution of complex queries, i.e., queries involving many relations. Notice that such complex queries can become frequent in real applications due to the use of views [28]. The architecture assumed in the performance study in Section 4 is a multiprocessor system with distributed memories and shared disks. 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. To facilitate our presentation and performance evaluation, the join method on which we shall demonstrate the application of hash filters is the sort-merge join that most existing database management softwares rely upon. Note that the concept of interleaving a bushy execution tree with hash filters is also applicable to improving the query execution time when other join methods, such as hash joins and nest-loop joins, are employed, and by no means confined to the use of sort-merge joins.

Both CPU and I/O costs of executing a query are considered. 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. By doing 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 utilizing hash filters in both cases. A detailed performance model on the cost of sort-merge joins and system parameters used is given in Section 4.

In addition, 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 [4]. In the presence of data skew [26], we only have to modify the corresponding formula accordingly [9].

3 Using Hash Filters for a Bushy Tree

In this section, we shall first evaluate the effect of hash filters and then propose a scheme to derive hash filters for a bushy execution tree.

3.1 The Effect of Hash Filters

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 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 since, as shown in Figure 1, different attribute

[2]Various heuristics, such as those in [6] and [17], can be applied to build a bushy execution tree. Note that assigning sequence numbers to joins while building a bushy tree involves little overhead.

values may be hashed into a same hash entry. To formally derive $\rho_{i,A}$, consider the ball drawing problem described below first.

Proposition 1: Suppose k balls are drawn sequentially and independently from m different balls. Then, the expected number of different balls selected is $m(1-(1-\frac{1}{m})^k)$.

It can be observed that 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 thus follows.

Proposition 2: 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.

Suppose R_j has two attributes A and B. The problem of estimating the cardinality of R_j projected on the non-filtered attribute B after $HF_{R_i}(A) \rightarrow R_j$ is very complicated, and needs to resort to the following combinatorial problem to resolve: "There are n balls with r different colors. Each ball has one color and the r colors are uniformly distributed over the n balls. Find the expected number of colors if h balls are randomly selected from the n balls." Denote the expected number of colors of the h selected balls as $g(r,n,h)$. Then, as pointed out in [27], $g(r,n,h)$ can be formulated as follows,

$$g(r,n,h) = r[1-\prod_{i=1}^{h}(\frac{\frac{n(r-1)}{r}-i+1}{n-i+1})]. \quad (2)$$

As shown in [2], Eq.(1) can be approximated as below,

$$g(r,n,h) \simeq \begin{cases} r, & \text{for } r < \frac{h}{2}, \\ h, & \text{for } h < \frac{r}{2}, \\ \frac{(r+h)}{3}, & \text{otherwise.} \end{cases} \quad (3)$$

We then obtain the reduction effect of a hash filter on a non-filtered attribute by assigning $|R_j| = n$, $|R_j(B)| = r$ and $|R_j|\rho_{i,A} = h$. It can be seen that when $|R_j(B)| = r$ is much less than $|R_j|\rho_{i,A} = h$, the cardinality of R_j(B) remains approximately the same after $HF_{R_i}(A) \rightarrow R_j$. Thus, we assume in this paper the number of distinct values of a non-filtered attribute remains the same after a hash filter application to simplify our discussion.

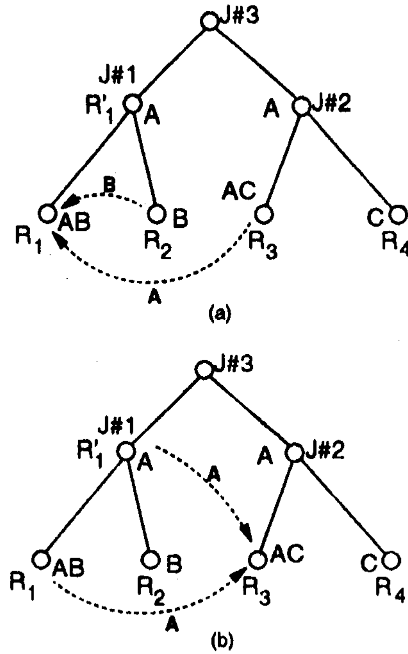


Figure 2: An example for the effect of hash filters.

As mentioned earlier, in a bushy tree execution, hash filters built in different execution stages can have very different reduction effects. To further investigate the effect of hash filters in a bushy tree, denote the set of relations within the subtree under R_i as $S(R_i)$. It can be seen that the size of an intermediate relation R_i will not be affected by the applications of hash filters between relations in $S(R_i)$. Consider the bushy tree in Figure 2a for example. Denote the resulting relation by $R_i \bowtie R_j$ as $R'_{\min\{i,j\}}$ for convenience. R'_1 in Figure 2a represents the resulting relation of join J#1. It can be verified that the application of $HF_{R_3}(A) \rightarrow R_1$ will reduce the size of R_1, and then that of R'_1. On the other hand, the application of $HF_{R_2}(B) \rightarrow R_1$ only reduces R_1, but not R'_1 since the effect of $HF_{R_2}(B)$ is offset by the join $R_1 \bowtie R_2$. This phenomenon can be stated by the proposition below.

Proposition 3: Suppose R_m is an intermediate relation in a bushy tree. The size of R_m will be reduced by $HF_{R_s}(A) \rightarrow R_d$ if and only if $R_d \in S(R_m)$ and $R_s \notin S(R_m)$.

Note that after a join, non-matched tuples are filtered out, meaning that $|R'_i(A)| \leq |R_i(A)|$ where $R'_i = R_i \bowtie R_j$. Thus, despite that the cardinality

of a resulting relation may be larger than those of its operands, the cardinality of distinct values of a certain attribute is always decreasing along the execution of a join sequence. This is the very reason that we shall generate hash filters based on the join sequence numbers to optimize their reduction effects in the algorithm to be described. For example, it can be seen that the reduction effect of $HF_{R'_1}(A) \rightarrow R_3$ is more powerful than that of $HF_{R_1}(A) \rightarrow R_3$ in Figure 2b. More formally, we have the following proposition for hash filters.

Proposition 4: $\rho_{i,A} \leq \rho_{j,A}$ if $R_j \in S(R_i)$.

3.2 Interleaving a Bushy Tree with HF's

In light of the results on the effect of hash filters in Section 3.1, we shall develop a scheme that applies hash filters to improving the execution of a bushy tree. The proposed scheme will interleave a given bushy tree with appropriate hash filters so that not only is the reduction effect optimized but also the cost is minimized. As pointed out earlier, the sort-merge join method is employed in our discussion on the use of hash filters. Let $\#J_{R_i}$ be the sequence number of the join which relation R_i is involved in. Joins with smaller sequence numbers execute first. R_i in $\#J_{R_i}$ can be either a base relation or an intermediate relation[3]. As can be seen from algorithm H below, the sequence number is used to determine the order of hash filters applied. Specifically, if $\#J_{R_i} < \#J_{R_j}$ and R_i and R_j have a common attribute A. Then R_j will build $HF_{R_j}(A)$ to apply to R_i. However, R_i does not build hash filter for R_j. Rather, in light of Proposition 4, the application of such a hash filter to R_j will be deferred until the execution reaches the ancestor of R_i, say R_k, such that $\#J_{R_k} \geq \#J_{R_j}$. The reduction effect by the hash filter on attribute A to R_j can thus be optimized.

Algorithm H: Interleaving a bushy tree with hash filters.

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

Step 2: **for** each leaf node R_i in T
    begin
        $S_{att} = \phi$;
        for each join attribute A of R_i
        Let R_j be the joining relation with R_i on attribute A.
        begin
            if $(\#J_{R_i} \geq \#J_{R_j})$ **then** $S_{att} = S_{att} \cup A$;
        end
        if $(S_{att} \neq \phi)$
        begin
            Scan R_i, and $\forall A \in S_{att}$, build $HF_{R_i}(A)$ by R_i;
            Send $HF_{R_i}(A)$ to R_j, where R_j is the joining relation with R_i on attribute A.
        end
    end

Step 3: **for** each leaf node R_i in T
    begin
        if R_i receives all HF's for its join attributes
        then
        begin
            R_i applies HF's to filter out non-matching tuples.
            R_i starts/resumes its sorting phase.
        end
    end

Step 4: **for** each join J in T
    begin
        if both relations R_i and R_j under J have completed their sorting phases **then**
        begin
            Perform the join J;
            (When generating the resulting relation R_s,)
            Generate $HF_{R_s}(A)$ for attribute A if $\exists$ a base relation R_y joining with R_s on A such that $\#J_{R_s} \geq \#J_{R_y}$;
            Send $HF_{R_s}(A)$ to its recipient;
            Update the execution tree T accordingly by removing R_i and R_j.
            (R_s becomes a leaf node.)
        end
    end

Step 5: **if** $|T|=1$ **then return results**
    else goto Step 3.

The operations of algorithm H can be described as follows. In Step 1, a bushy tree is built first. Then, relations involved in later joins will build hash filters for those involved in earlier joins in Step 2. Let S_{att} be the set of attributes to build HF's. The first conditional statement in Step 2 to set up S_{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 S_{att}. Every relation, after receiving and utilizing all its filters, starts its sorting phase in Step 3. The merge phase of a join begins when all of its operands are available in Step 4. It can be observed that building HF's can be carried out when output tuples are being generated, thus avoiding another relation scan. The procedure repeats until all joins are completed as stated in Step 5.

[3] In the case of dealing with a segmented right-deep tree, which is a bushy tree with right-deep subtrees [4], one can use segment sequence numbers, instead of join sequence numbers, to properly insert hash filters into the bushy tree among different segments.

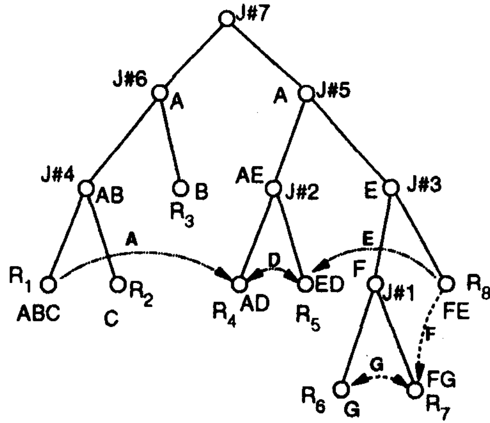


Figure 3: Application of hash filters for joins J#1 and J#2.

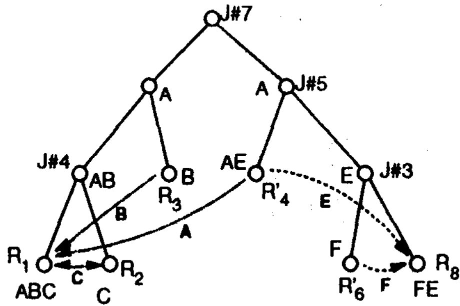


Figure 4: Application of hash filters for J#3 and J#4.

3.3 Examples and Variations

Consider the bushy tree in Figure 3 for example. Since $R_6 \bowtie R_7$ is the first join to perform, we have $HF_{R_8}(F) \rightarrow R_7$, $HF_{R_6}(G) \rightarrow R_7$ and $HF_{R_7}(G) \rightarrow R_6$ before the execution of $R_6 \bowtie R_7$. Then, prior to the second join $R_4 \bowtie R_5$, four hash filters, $HF_{R_8}(E) \rightarrow R_5$, $HF_{R_1}(A) \rightarrow R_4$, $HF_{R_4}(D) \rightarrow R_5$ and $HF_{R_5}(D) \rightarrow R_4$ are applied. The bushy tree after the first two joins is shown in Figure 4. We in turn have the hash filters $HF_{R'_4}(E) \rightarrow R_8$ and $HF_{R'_6}(F) \rightarrow R_8$ applied as shown in Figure 4 before the join $R'_6 \bowtie R_8$. Similarly, following the operations in algorithm H, the applications of hash filters are illustrated in Figures 4 and 5. It can be seen that to have a better reduction effect according to Proposition 4, $HF_{R'_4}(A) \rightarrow R_1$ and $HF_{R'_4}(E) \rightarrow R_8$ are built after the join $R_4 \bowtie R_5$, instead of being built by R_4 and R_5, respectively, in the bushy tree in Figure 3.

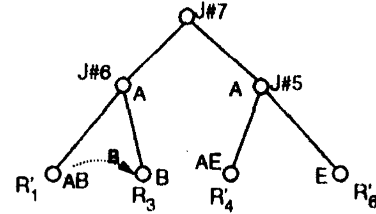


Figure 5: Application of hash filters for J#5, J#6 and J#7.

Clearly, there are many variations of algorithm H above. To provide more insights into the approach of hash filters, extensive simulation will be conducted in Section 4 to evaluate various schemes using hash filters. For notational readability, algorithm H will be denoted by CA in what follows, where CA stands for its nature of "check and apply." Instead of interleaving the joins in a bushy tree with hash filters, hash filters can be built directly from base relations and applied as a preprocessing of a bushy tree. Such an approach will be referred to as scheme SM, where SM stands for "simple." Also, hash filters can be regenerated from intermediate relations, and repeatedly applied to achieve better reduction effect at the cost of employing more hash filters. This alternative is denoted by RG, standing for "regeneration." The conventional approach without using hash filters, denoted by NF (i.e., "no hash filters"), will also be implemented for a comparison purpose.

Note that the first step of the sorting phase can be performed while a hash filter is being built to minimize both CPU and I/O costs. In addition, in the case that indices are available for certain attributes, we can scan the corresponding indices instead of the whole relation in Step 2 to reduce the cost. Optimization on these issues is system dependent, and can in fact further increase the performance improvement achievable by using hash filters.

4 Performance Study

We first describe the performance model used to evaluate the benefit of different hash filter generation and application schemes in Section 4.1. Parameters used in simulation are given in Section 4.2. Simulation

results are then presented and analyzed in Section 4.3.

4.1 Model Overview

The performance model consists of three major components: Query Manager, Optimizer, and Executor. Query Manager 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.

Optimizer takes a query request from Query Manager and produces a query plan in the form of a bushy tree. Join sequence numbers are assigned to internal nodes of the bushy tree to represent the order of join operations determined by Optimizer. The bushy tree query plan is determined by the ***minimum cost*** heuristic described in [6] that tries to perform the join with the minimal cost first.

Executor traverses the query plan tree and carries out join operations sequentially according to join sequence numbers determined by Optimizer. As mentioned earlier, the sort-merge join method is used. Depending upon the schemes simulated, hash filters of join attributes are generated at different stages of query execution. Note that unlike those hash filters in SM and CA that can only be applied to base relations, those in RG can even be applied to intermediate relations.

Our model computes both CPU and I/O costs of executing a query. CPU cost for sorting and merging is determined by the total number of tuples processed multiplied by the number of CPU instructions per tuple. We assume that the costs of sorting and merging for each tuple are the same, and both are equal to I_{tuple}. Using sort-merge joins, it takes $O(N \log N)$ steps to sort a relation with N tuples, and takes from $O(N_1 + N_2)$ to $O(N_1 \times N_2)$ steps to merge two sorted relations of size N_1 and N_2. Under the assumption that attribute values are uniformly distributed over the attribute domain, the CPU cost of joining two relations in our model can be approximated as $I_{tuple} \times (N_1 \log N_1 + N_2 \log N_2 + N_1 + N_2)$. The CPU processing time is obtained by dividing the total number of CPU instructions per query by the CPU speed, CPU_{speed}. By dealing with the pathlength per tuple and the CPU speed, we can vary the CPU speed to make a query execution either CPU bound or I/O bound, and study the impact of using hash filters in both cases.

I/O cost for processing a query is determined by disk service time per page, t_{pio}, multiplied by the total number of page reads and writes. To sort a relation of P pages, $\log_m P + 1$ iterations of disk I/O are required, where m is the number of main memory buffer pages available for sorting. Each iteration reads P pages into memory for sorting and writes P sorted pages to disk. To merge two sorted relations of P_1 and P_2 pages, $P_1 + P_2$ pages are read into memory. The number of pages written to disk after a join operation is determined by the size of the resulting relation, P_r. Thus, the total number of I/O's required to join two relations of size P_1 and P_2 is $2 \times (P_1(\log_m P_1 + 1) + P_2(\log_m P_2 + 1)) + P_r$.

CPU cost for generating and applying HF's is determined by two parameters, I_{hash} and I_{probe}. I_{hash} is the number of CPU instructions required to generate hash value and set the corresponding bit in the hash filter for each tuple. I_{probe} 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. The CPU cost of generating HF for a join attribute is computed by multiplying I_{hash} by the relation cardinality. Note that HF generation phase can be combined with the first step of the sorting phase of a join, thus avoiding I/O overhead for HF generation. CPU cost for applying an HF is equal to I_{probe} multiplied by the relation cardinality. Also, 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.

4.2 Parameter Setting

To simplify our simulation study, we assume that join operations in a bushy tree are executed sequentially, thus not resorting to inter-operator parallelism to demonstrate the power of hash filters. The impact of combining the use of hash filters and parallel query execution is slated for future study. We select queries of four sizes, i.e., queries with 4, 8, 12, and 16 relations.

parameters	setting
I_{tuple}	300
I_{hash}	100
I_{prob}	200
m	2K pages
p_{size}	40 tuples
t_{pio}	15 ms
R_{card}	1M tuples
A_{card}	700K
carv	100K - 600K
attv	100K - 400K
$f_d(A)$	uniform
$f_d(R)$	uniform
CPU_{speed}	2 - 10 MIPS

Table 1: Parameters used in simulation.

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, [7], [12], and [21] were referenced to determine the values of simulation parameters. Table 1 summarizes the parameter settings used in simulation. The number of CPU instructions per tuple read was set to 300 while those for HF generation and application are set to 100 and 200, respectively. The buffer was assumed to have 2K pages, and each page was assumed to contain 40 tuples. Disk service time per page was assumed to be 15 milliseconds while the CPU speed was set to either 2 MIPS or 10 MIPS.

4.3 Simulation Results

In the simulation program, which was coded in C, the action for each individual relation to go through join operations, as well as generate and apply hash filters, was simulated. For each query in the simulation, four schemes, i.e., NF (no filter), SM (simple), CA (check and apply) and RG (regenerate HF), were applied to execute the query, and the execution time for each scheme was obtained.

Experiment 1: 10 MIPS CPU with attv= 100K and carv= 100K

In the first experiment, the CPU speed was set to 10 MIPS while both *attv* and *carv* were set to 100K. The average CPU, I/O, and total costs for this experiment are shown in Figures 6, 7, and 8, respectively. In these figures and all following figures except Figure 12, the ordinate is the execution time in seconds while the abscissa denotes the number of relations in a query. Figures 6 and 7 show that with 10 MIPS CPU, these queries using the sort-merge join method are I/O bound. The 15 ms page I/O time setting assumes sequential I/O without prefetching or disk buffering (e.g., reading one track at a time). Note that this experiment could become CPU bound if disk buffering or a larger page size was used.

Using the sort-merge join method, the I/O cost of sorting a relation of P pages is of the order $t_{pio} \times P \times \log_m P$, while the CPU cost is of the order $t_{tuple} \times R_{card} \times \log R_{card}$, where t_{tuple} is the sorting time per tuple ($\approx I_{tuple}/CPU_{speed}$) and R_{card} is equal to $P \times p_{size}$. Given the parameter settings in Table 1, the I/O cost for sorting two 1M tuple relations is approximately equal to 1,000 seconds while the corresponding CPU cost is approximately 1,200 seconds. I/O cost for merging two sorted relations is about 750 seconds, plus the I/O cost of writing the resulting relation to disk, whereas the CPU cost associated is about 60 seconds. This accounts for the reason that Experiment 1 is I/O bound.

Figures 6 and 7 also show that using hash filters results in a slight performance improvement in terms of both CPU and I/O costs required when *sn* is small (sn $\leq$ 8). The improvement increases significantly as the number of relations increases. It can be seen from Figure 8 that CA performs the best among all schemes evaluated while NF is outperformed by all other schemes. As described in Section 3, CA is devised with the goal of optimizing the reduction effect of HF's as well as minimizing the cost associated. The results from this experiment confirm our analysis in Section 3. Note that SM performs better than RG when $sn \leq 12$, while the latter performs better when $sn = 16$. This can be explained as follows. First, the additional filtering (size reduction) effect by applying a hash filter generated by an intermediate relation (say R_i) to relation R_j under RG is usually not significant if a hash filter on the same attribute has been generated by a offspring of R_i and applied to R_j, or a offspring of R_j, before. Second, RG consumes extra system resources to regenerate HF's after every join operation, except the last one. When *sn* is small, the cost of generating additional HF's is larger than the benefit of additional size reduction. When *sn* increases, the depth of the query execution tree increases, which in

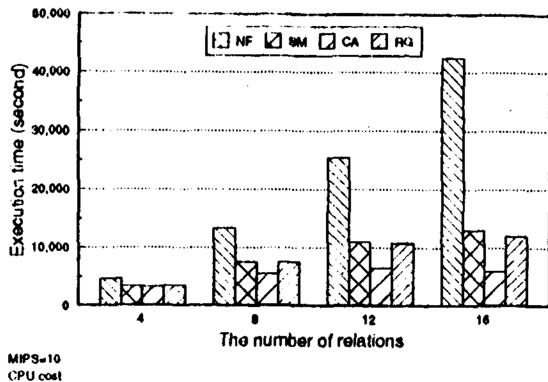


Figure 6: The CPU cost for each scheme when MIPS=10.

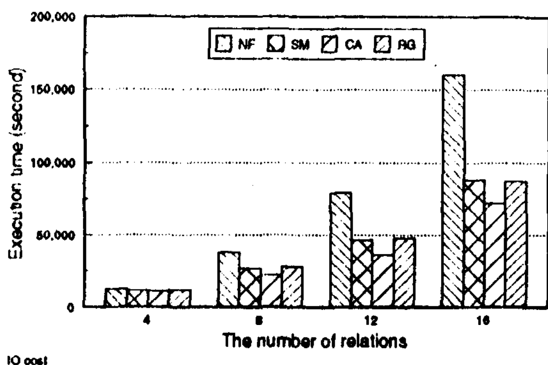


Figure 7: The IO cost for each scheme.

turn causes more join operations to benefit from the effect of additional filtering. As a result, the benefit provided by additional filtering in RG outweighs the cost of additional HF generations when *sn* is large.

The minimum, maximum, and standard deviation of query execution time for the four schemes when *sn*= 12 are shown in Table 2. The standard deviation of the query execution time is about 7.9% of mean for NF, whereas those are 18.9%, 26.2%, and 25% of mean for SM, CA, and RG, respectively. Note that the *minimum cost* heuristic used by our model to determine the bushy execution tree does not consider the effect of hash filters. Thus, the benefits of using hash filters in different bushy trees vary. This is the very reason that SM, CA, and RG produce larger relative standard deviations than NF.

The number of hash filters applied in each scheme is shown in Table 3. SM and CA apply the same number of hash filters for each query, since in both schemes, HF's are applied to base relations only. In

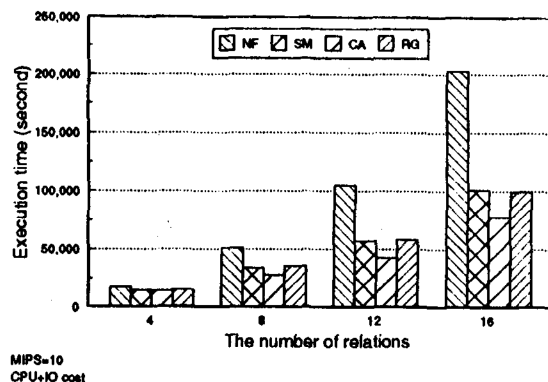


Figure 8: The total cost for each scheme when MIPS=10.

	standard dev	maximum	minimum
NF	8306	149234	92496
SM	10900	99012	38631
CA	11280	91901	26977
RG	14704	114385	35681

Table 2: Statistics for the cost of each scheme when the query size is 12 and MIPS=10.

RG, in addition to HF's applied to base relations, a hash filter for the next join attribute is regenerated from the resulting relation after every join. RG therefore generates and applies the most hash filters. However, our simulation results show that RG performs worse than both CA and SM when *sn* is small ($sn \leq 12$). As previously explained, this is due to the fact that the effect of hash filters diminishes as they are repeatedly applied, and is thus not worthwhile the cost of generating additional hash filters. This indeed agrees with the estimation in Eq.(3), which states that the number of distinct values of a non-filtered attribute only slightly decreases after the application of a hash filter. When *sn* is large ($sn > 12$), RG performs better than SM, but still worse than CA.

no. of relations	4	8	12	16
SM	6	18	32	48
CA	6	18	32	48
RG	8	24	42	62

Table 3: The average number of hash filters applied in each scheme.

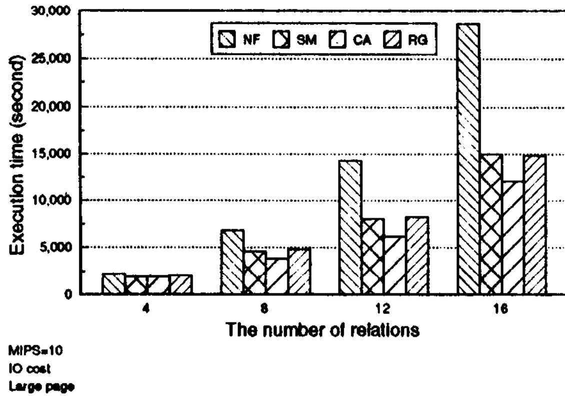


Figure 9: The IO cost for each scheme.

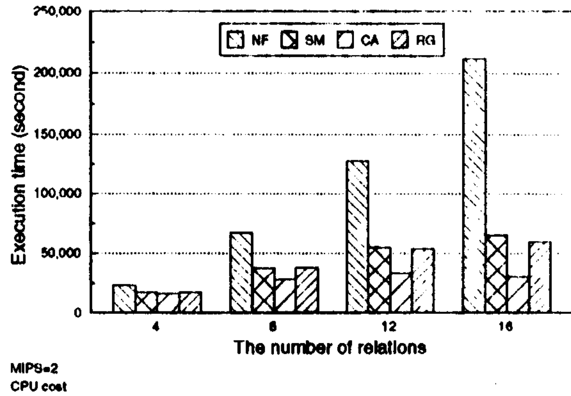


Figure 10: The CPU cost for each scheme when MIPS=2.

As pointed out earlier, the above experiment can become CPU bound if the disk access time is reduced. To provide more insights into this phenomenon, an experiment is conducted, where the page size is increased to 480 tuples, approximately equal to the track size of a typical workstation disk nowadays. Disk access time per page thus increases to 30 ms accordingly while all other parameters remain unchanged. The average I/O costs for the four schemes in this experiment are shown in Figure 9. Note that since CPU speed remains as 10 MIPS, CPU costs for the four schemes are the same as those in Figure 6. From Figures 7 and 9, it can be seen that I/O costs for the four schemes in this experiment are significantly reduced as compared to those required in the prior experiment. Consequently, this experiment is CPU bound as evidenced by the results in Figures 6 and 9.

Experiment 2: 2 MIPS CPU with attv= 100K, and carv= 100K

In Experiment 2, the CPU speed was changed to 2 MIPS while all other parameters remained the same as in Experiment 1. The average CPU cost for this experiment is shown in Figure 10. Since changing the CPU speed does not affect I/O costs, I/O costs for the four schemes in this experiment are the same as those in Experiment 1, as shown in Figure 7. It can be seen from Figures 7 and 10 that queries in Experiment 2 are CPU bound under NF. Figures 6 and 10 show that the three HF based schemes lead to larger reductions on CPU cost when queries are CPU bound, but their relative improvement over NF is approximately the same in both experiments. Figure 11 shows the average query execution times (i.e, CPU cost + I/O cost) for the four schemes. It can be observed that relative performance among these schemes is very similar to that in Experiment 1. CA continues to outperform the other three schemes while NF still performs the worst. The three schemes utilizing hash filters reduce the query execution time of NF by more than 50%, when $sn \geq 12$.

The improvement of CA over NF for both Experiments 1 and 2 is shown in Figure 12, where the ordinate is the ratio of execution time of CA to NF, and the abscissa denotes the number of relations in a query. It can be seen from Figure 12 that the improvement increases as sn increases. When $sn = 4$, the execution of CA is about 84% of that of NF with 10 MIPS CPU, and this ratio becomes 76% with 2 MIPS CPU. When $sn = 16$, such a ratio decreases to about 39% with 10 MIPS CPU, and to 28% with 2 MIPS CPU. Figure 12 also shows that CA generates a larger cost reduction when queries are CPU bound. Note that with a slower CPU, the absolute CPU cost reduction achieved by CA is larger. Since the I/O cost is not affected by the change in CPU speed, the ratio of cost reduction by CA becomes larger when CPU is slower. Experiments 1 and 2 demonstrate that hash filter is a very powerful means to reduce the query execution time, especially for complex queries, in both CPU and I/O bound cases.

The minimum, maximum, and standard deviation of query execution time for each scheme with sn=12 are shown in Table 4, where CA again has the smallest maximum and minimum execution times, but the second largest standard deviation, agreeing with our observation in Experiment 1.

5 Conclusions

In this paper, we explored an approach of interleaving a bushy execution tree with hash filters to improve the

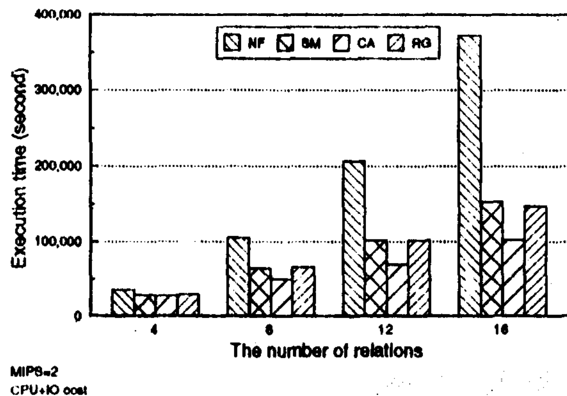


Figure 11: The total cost for each scheme when MIPS=2.

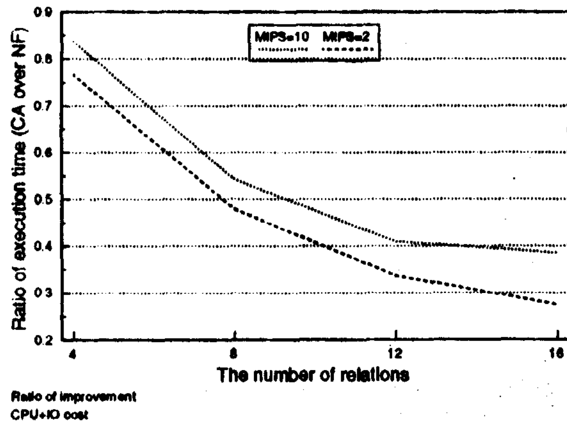


Figure 12: Execution cost ratio of CA to NF.

	standard dev	maximum	minimum
NF	15632	293295	184366
SM	20028	180580	67206
CA	21485	165182	41077
RG	27389	206659	60207

Table 4: Statistics for the cost of each scheme when sn=12 and MIPS=2.

execution of multi-join queries. An efficient scheme to determine an effective sequence of hash filters for a bushy execution tree has been developed, where the hash filters are built and applied based on the join sequence specified in the bushy tree so that not only is the reduction effect optimized but also the cost associated is minimized. Various schemes using hash filters were implemented and evaluated via simulation. By varying the CPU speed, both CPU and I/O bound jobs were investigated. Extensive simulation results were obtained to provide insights into the use of hash filters. It is experimentally shown that the application of hash filters is in general a very powerful means to improve the execution of multi-join queries, and the improvement becomes more prominent as the number of relations in a query increases.

References

[1] E. Babb. Implementing a Relational Database by Means of Specialized Hardware. *ACM Transactions on Database Systems*, 4(1):1–29, March 1979.

[2] P. A. Bernstein and D.-M. W Chiu. Using Semi-Joins to Solve Relational Queries. *Journal of ACM*, 28(1):25–40, January 1981.

[3] 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.

[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 and P. S. Yu. Interleaving a Join Sequence with Semijoins in Distributed Query Processing. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):611–621, September 1992.

[6] 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.

[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] 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.

[10] W. Hong. Exploiting Inter-Operator Parallelism in XPRS. *Proceedings of ACM SIGMOD*, pages 19–28, June 1992.

[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] 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.

[14] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of Nonrecursive Queries. *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 128–137, August 1986.

[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] R. A. Lorie, J.-J. Daudenarde, J. W. Stamos, and H. C. Young. Exploiting Database Parallelism In A Message-Passing Multiprocessor. *IBM Journal of Research and Development*, 35(5/6):681–695, September/November 1991.

[17] 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.

[18] P. Mishra and M. H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[19] H. Pirahesh, C. Mohan, J. Cheng, T. S. Liu, and P. Selinger. Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches. *Proceedings of the 2nd International Symposium on Databases in Parallel and Distributed Systems*, pages 4–29, July 1990.

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

[21] 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.

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

[24] A. Swami and A. Gupta. Optimization of Large Join Queries. *Proceedings of ACM SIGMOD*, pages 8–17, 1988.

[25] P. Valduriez and G. Gardarin. Join and Semijoin Algorithms for a Multiprocessor Database Machine. *ACM Transactions on Database Systems*, 9(1):133–161, March 1984.

[26] J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek. An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew. *Proceedings of the 7th International Conference on Data Engineering*, pages 200–209, April 1991.

[27] S. B. Yao. Approximating block access in database organizations. *Comm. of ACM*, 20:260–261, April 1977.

[28] P. S. Yu, M.-S. Chen, H. Heiss, and S. H. Lee. On Workload Characterization of Relational Database Environments. *IEEE Transactions on Software Engineering*, 18(4):347–355, April 1992.