The VLDB Journal (1997) 6: 121–131

The VLDB Journal
© Springer-Verlag 1997

# On applying hash filters to improving the execution of multi-join queries

**Ming-Syan Chen**[1], **Hui-I Hsiao**[2], **Philip S. Yu**[2]

[1] Electrical Engineering Department, National Taiwan University, Taipei, Taiwan
[2] IBM T.J. Watson Research Center, P.O.Box 704, Yorktown, NY 10598, USA

Edited by G. Gardarin. Received October 1994 / Accepted December 1995

**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. Similar to semi-joins in distributed query processing, hash filters can be applied to eliminate non-matching tuples from joining relations before the execution of a join, thus reducing the join cost. Note that hash filters built in different execution stages of a bushy tree can have different costs and effects. 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.

**Key words:** Hash filters – Parallel query processing – Bushy trees – Sort-merge joins

## 1 Introduction

Parallel database machines have drawn a considerable amount of attention from both the academic and industrial communities due to their high potential for parallel execution of complex database operations [4, 8, 16, 27, 30]. In relational database systems, joins are the most expensive operations to execute, especially with the increases in database size and query complexity [15, 21, 34]. Database applications which involve decision support and complex objects usually have to specify their desired results in terms of multi-join queries, and some complex queries for such applications may take hours or even days to complete, thus degrading the system performance. As a result, it has become imperative to develop solutions for efficient execution of multi-join queries for future database management [9, 11, 19, 22].

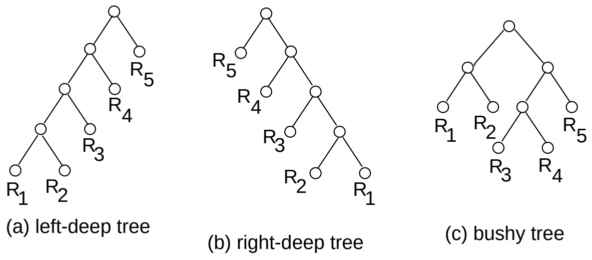


**Fig. 1a–c.** Illustration of different query trees

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. Examples of the three forms of query trees are shown in Fig. 1. A significant amount of research efforts has been elaborated upon developing join sequences to improve the query execution time. The work reported in [26] 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 [12, 17, 28, 29].

On the other hand, the bushy tree join sequences 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 fact that in the past the power/size of a multi-processor 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 multi-processors and the trend for queries to become more complicated [34], 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 [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 [20], where a greedy scheme taking various join methods and their corresponding costs into consideration was proposed. As an extension to [12], an algorithm-handling processor scheduling in a bushy tree was proposed in [11], 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 [11]. A two-step approach to deal with join sequence scheduling and processor allocation for parallel query processing using sort-merge joins was devised in [7]. Pipelining hash joins in a bushy tree and processor allocation within each pipeline were studied in [5] and [18], respectively. In addition, various query plans in processing multi-join queries in a shared-nothing architecture were investigated in [24].

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 semi-join has traditionally been relied upon to reduce the amount of inter-site data transmission required for distributed query processing [2, 6], the technique of hash filtering can be applied in a parallel database environment to reduce the relation cardinalities. Note, however, that previous 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, 8, 23, 31][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 (HFs) 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. Due to the complexity of a bushy tree, HFs built and applied in different execution stages can have very different costs and reduction effects. How to build HFs 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. To the best of our knowledge, despite its importance, there is no previous work on exploring the approach of interleaving a bushy execution tree with HFs to improve the execution of multi-join queries, let alone conducting studies of its performance. This feature distinguishes our work from that of others.

[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 |

(a)

$R_2$

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

| 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)

**Fig. 2a,b.** An example of the use of HFs.

An 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 zeros. 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 semi-joins, 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 HFs can be found in Fig. 2, where an $HF_{R_1}(B)$ is built by $R_1$ and applied to $R_2$, with the corresponding hash function $h(b_i)$= $i$ mod 5. It can be verified that, after the application of $HF_{R_1}(B)$, $R_2$ is reduced to the one given in Fig. 2b, thus reducing the join cost of $R_1 \bowtie R_2$. Note that the effect of HFs is more complicated than that of semi-joins, since hash collision can occur for different attribute values (such as $b_1$ and $b_6$ in Fig. 2a) when an HF is built. In this paper, we shall evaluate the effect of HFs first, and then develop an efficient scheme to interleave a bushy execution tree with HFs to minimize the query execution cost. As mentioned earlier, HFs 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, HFs 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 HFs via simulation. It is experimentally shown that the application of HFs 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.

[2] Various heuristics, such as those in [7] and [20], can be applied to build a bushy execution tree. Note that assigning sequence numbers to joins while building a bushy tree involves little overhead

The rest of this paper is organized as follows. Preliminaries are given in Sect. 2. The effect of HFs and the proposed scheme are presented in Sect. 3. Performance studies on various schemes using HFs are conducted in Sect. 4 via simulation. The paper concludes with Sect. 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 previous 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 [34]. The architecture assumed in the performance study in Sect. 4 is a multiprocessor system with distributed memories and shared disks containing database data. Barring any tuple placement skew [32], 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 HFs is the sort-merge join that most existing database management softwares rely upon. Note that the concept of interleaving a bushy execution tree with HFs 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 path length, 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/Os. 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 HFs in both cases. A detailed performance model on the cost of sort-merge joins and the system parameters used is given in Sect. 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 previous work [7] that is given in the Appendix for reference[3]. In the presence of data skew, we only have to modify the corresponding formula accordingly [10].

## 3 Using HFs for a bushy execution tree

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

### *3.1 The effect of HFs*

Let $HF_{R_i}(A)\rightarrow R_j$ denote the application of an HF generated by $R_i$ on attribute $A$ 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 an HF since, as shown in Fig. 2, different attribute 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 $m(1-(1-\frac{1}{m})^k)$.*

*Proof.* Let $X_i$=1 if the i-th ball is drawn at least once, and $X_i$=0 otherwise. $S = \sum_{i=1}^{m} X_i$ is the number of distinct balls drawn. Then $E(S)= \sum_{i=1}^{m} E\{X_i\}= mE\{X_i\}= m(1-(1-\frac{1}{m})^k)$. **Q.E.D.**

It can be observed that hashing $k = |R_i(A)|$ different values into an HF of $m$ bits is similar to the experiment of drawing $k$ balls from $m$ different balls with replacement. The following proposition thus follows.

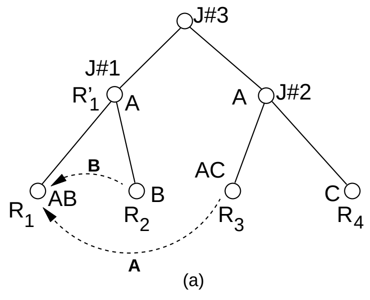
**Proposition 2.** *The reduction ratio by the application of $HF_{R_i}(A)$, $\rho_{i,A}$, can be formulated as*

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

*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 an HF.*

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 [33], $g(r, n, h)$ can be formulated as follows,

[3] Note that this formula offers a more sophisticated model than the one based on the foreign key assumption



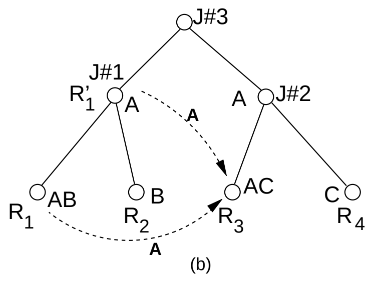


**Fig. 3a,b.** An example of the effect of HFs

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

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

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

We then obtain the reduction effect of an HF 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 an HF application to simplify our discussion.

As mentioned earlier, in a bushy tree execution, HFs built in different execution stages can have very different reduction effects. To further investigate the effect of HFs 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 HFs between relations in $S(R_i)$. Consider the bushy tree in Fig. 3a for example. Denote the resulting relation by $R_i \bowtie R_j$ as $R'_{\min\{i,j\}}$ for convenience. $R'_1$ in Fig. 3a 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 $|R'_i(A)| \le |R_i(A)|$ where $R'_i = R_i \bowtie R_j$. Thus, despite the cardinality of a resulting relation possibly being 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 HFs 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 Fig. 3b. Formally, we have the following proposition for HFs.

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

### *3.2 Interleaving a bushy execution tree with HFs*

In light of the results on the effect of HFs in Sect. 3.1, we shall develop a scheme that applies HFs to improving the execution of a bushy tree. The proposed scheme will interleave a given bushy tree with appropriate HFs, 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 HFs. Let $\#J_{R_i}$ be the sequence number of the join in which relation $R_i$ is involved. Joins with smaller sequence numbers execute first. $R_i$ in $\#J_{R_i}$ can be either a base relation or an intermediate relation[4]. As can be seen from algorithm $H$ below, the sequence number is used to determine the order of HFs 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 an HF for $R_j$. Rather, in light of Proposition 4, the application of such an HF to $R_j$ will be deferred until the execution reaches the ancestor of $R_i$, say $R_k$, such that $\#J_{R_k} \ge \#J_{R_j}$. The reduction effect by the HF on attribute $A$ to $R_j$ can thus be optimized.

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 HFs for those involved in earlier joins in Step 2. Let $S_{\text{att}}$ be the set of attributes to build HFs. The first conditional statement in Step 2 to set up $S_{\text{att}}$ assures that only necessary HFs will be generated and applied to other relations. Also, it can be seen that a relation will be scanned at most once to build HFs for attributes in $S_{\text{att}}$. 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 HFs 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.

```
Algorithm $H$: Interleaving a bushy execution tree with HFs.
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**
    $S_{\text{att}} = \phi$;
    **for** each join attribute $A$ of $R_i$
```

[4] In the case of dealing with a segmented right-deep tree, which is a bushy tree with right-deep subtrees [5], one can use segment sequence numbers, instead of join sequence numbers, to properly insert HFs into the bushy tree among different segments

```
        Let $R_j$ be the joining relation with $R_i$ on attribute $A$.
        begin
            if (#$J_{R_i} \geq$ #$J_{R_j}$) then $S_{\text{att}}$= $S_{\text{att}} \cup A$;
        end
        if ($S_{\text{att}} \neq \phi$)
        begin
            Scan $R_i$, and $\forall$ $A \in S_{\text{att}}$, 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 HFs for its join attributes then
        begin
            $R_i$ applies HFs 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.
```

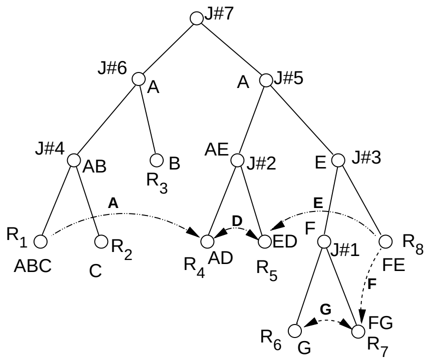


**Fig. 4.** Application of HFs for joins J#1 and J#2

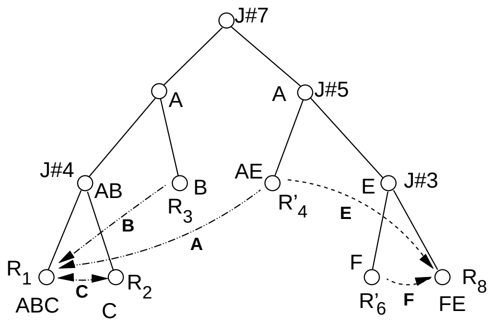


**Fig. 5.** Application of HFs for joins J#3 and J#4

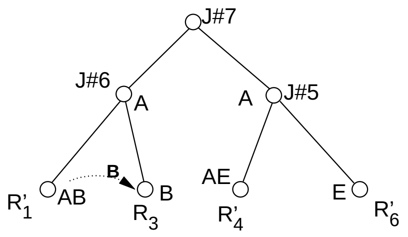


**Fig. 6.** Application of HFs for joins J#5, J#6 and J#7

*3.3 Examples and variations*

Consider the bushy tree in Fig. 4 for example. Since $R_6 \bowtie R_7$ is the first join to perform, we have $HF_{R_8}(F) \to R_7$, $HF_{R_6}(G) \to R_7$ and $HF_{R_7}(G) \to R_6$ before the execution of $R_6 \bowtie R_7$. Then, prior to the second join $R_4 \bowtie R_5$, four HFs, $HF_{R_8}(E) \to R_5$, $HF_{R_1}(A) \to R_4$, $HF_{R_4}(D) \to R_5$ and $HF_{R_5}(D) \to R_4$ are applied. The bushy tree after the first two joins is shown in Fig. 5. We, in turn, have the HFs $HF_{R'_4}(E) \to R_8$ and $HF_{R'_6}(F) \to R_8$ applied as shown in Fig. 5 before the join $R'_6 \bowtie R_8$. Similarly, following the operations in algorithm $H$, the applications of HFs are illustrated in Figs. 5 and 6. It can be seen that to have a better reduction effect according to Proposition 4, $HF_{R'_4}(A) \to R_1$ and $HF_{R'_4}(E) \to R_8$ 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 Fig. 4.

Clearly, there are many variations of algorithm $H$. To provide more insights into the approach of HFs, extensive simulation will be conducted in Sect. 4 to evaluate various schemes using HFs. For notational readability, algorithm $H$ will be denoted in the following by CA, where CA stands for its nature of "check and apply." Instead of interleaving the joins in a bushy tree with HFs, the latter 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, HFs can be regenerated from intermediate relations, and repeatedly applied to achieve better reduction effect at the cost of employing more HFs. This alternative is denoted by RG, standing for "regeneration." The conventional approach without using HFs, denoted by NF (i.e., "no filters"), will also be implemented for a comparison purpose.

Note that the first step of the sorting phase can be performed while an HF 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 HFs.

## 4 Performance study

We first describe the performance model used to evaluate the benefit of different HF generation and application schemes in Sect. 4.1. Parameters used in simulation are given in Sect. 4.2. Simulation results are then presented and analyzed in Sect. 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: $R_{\text{card}}$, $carv$, $f_d(R)$, $A_{\text{card}}$, $attv$, and $f_d(A)$. Relation cardinalities in a query are computed from a distribution function, $f_d(R)$, with a mean, $R_{\text{card}}$, and a deviation, $carv$. Cardinalities of join attributes are determined similarly by $A_{\text{card}}$, $attv$, 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 [7] 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, HFs of join attributes are generated at different stages of query execution. Note that unlike those HFs 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 $I_{\text{tuple}}$. 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 $I_{\text{tuple}} \times (N_1 \log N_1 + N_2 \log N_2 + N_1 + N_2)$. The CPU processing time is obtained by dividing the total number of CPU instructions per query by the CPU speed, $CPU_{\text{speed}}$. By dealing with the path length 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 HFs in both cases.

**Table 1.** Parameters used in simulation

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

I/O cost for processing a query is determined by disk service time per page, $t_{\text{pio}}$, 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/Os 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 HFs is determined by two parameters, $I_{\text{hash}}$ and $I_{\text{probe}}$. $I_{\text{hash}}$ is the number of CPU instructions required to generate hash value and set the corresponding bit in the HF for each tuple. $I_{\text{probe}}$ 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 an HF for a join attribute is computed by multiplying $I_{\text{hash}}$ by the relation cardinality. Note that the 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 $I_{\text{probe}}$ multiplied by the relation cardinality. Also, in our simulation model, HFs are implemented as bit-vectors and can in general fit in memory, thus minimizing extra I/Os 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 HFs. The impact of combining the use of HFs 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. This set of selections covers a wide spectrum of query sizes ranging from a simple three-way join to a more than 20-way join. For each query size, 500 query graphs were generated, and, as mentioned in Sect. 4.1, only queries with connected query graphs are used in our study.

To conduct the simulation, [3], [8], [13], and [25] were referenced to determine the values of simulation parameters.

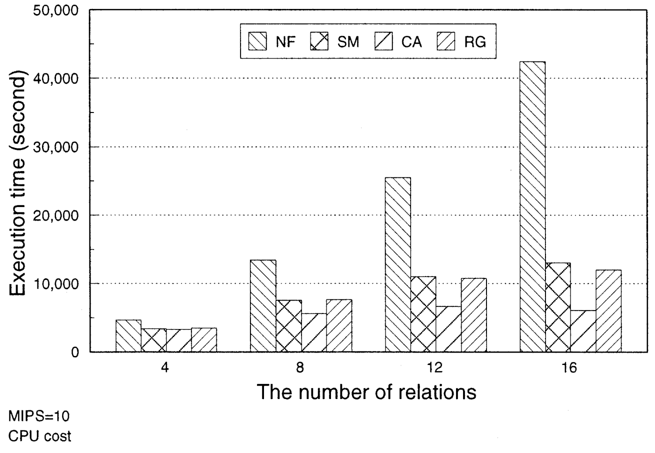


**Fig. 7.** The CPU cost for each scheme when MIPS=10

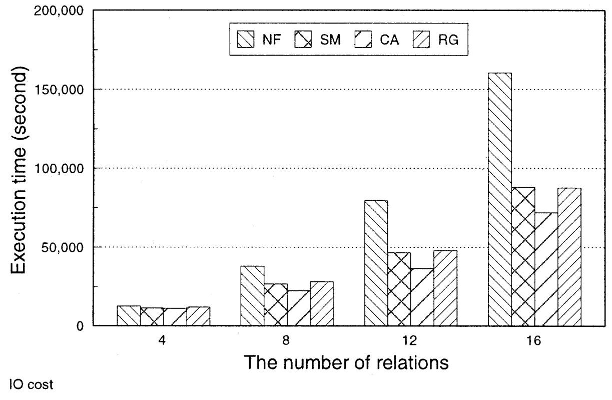


**Fig. 8.** The I/O cost for each scheme

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 ms 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 HFs, 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.

#### 4.3.1 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 Figs. 7, 8, and 9, respectively. In these figures and all following figures except Fig. 13, the ordinate is the execution time in seconds while the abscissa denotes the number of relations in a query. Figures 7 and 8 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.

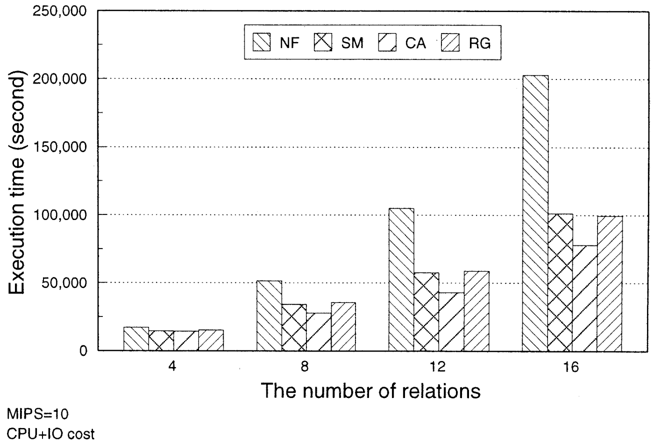


**Fig. 9.** The total cost for each scheme when MIPS=10

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

Figures 7 and 8 also show that using HFs 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 Fig. 9 that CA performs the best among all schemes evaluated, while NF is outperformed by all other schemes. As described in Sect. 3, CA is devised with the goal of optimizing the reduction effect of HFs as well as minimizing the cost associated. The results from this experiment confirm our analysis in Sect. 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 an HF generated by an intermediate relation (say $R_i$) to relation $R_j$ under RG is usually not significant if an HF 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 HFs after every join operation, except the last one. When $sn$ is small, the cost of generating additional HFs is larger than the benefit of additional size reduction. When $sn$ increases, the depth of the query execution tree increases, which in turn causes more join operations to benefit from the effect

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

| | Standard dev | Maximum | Minimum |
|---|---|---|---|
| NF | 8306 | 149234 | 92496 |
| SM | 10900 | 99012 | 38631 |
| CA | 11280 | 91901 | 26977 |
| RG | 14704 | 114385 | 35681 |

**Table 3.** The average number of HFs applied in each scheme

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

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 HFs. Thus, the benefits of using HFs 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 HFs applied in each scheme is shown in Table 3. SM and CA apply the same number of HFs for each query, since in both schemes, HFs are applied to base relations only. In RG, in addition to HFs applied to base relations, an HF for the next join attribute is regenerated from the resulting relation after every join. RG therefore generates and applies the most HFs. 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 HFs diminishes as they are repeatedly applied, and is thus not worthwhile the cost of generating additional HFs. 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 an HF. When $sn$ is large ($sn > 12$), RG performs better than SM, but still worse than CA.

As pointed out earlier, the above experiment can become CPU bound if the disk access time is reduced. To provide more insight 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 Fig. 10. Note that since CPU speed remains at 10 MIPS, CPU costs for the four schemes are the same as those in Fig. 7. From Figs. 8 and 10, 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 Figs. 7 and 10.

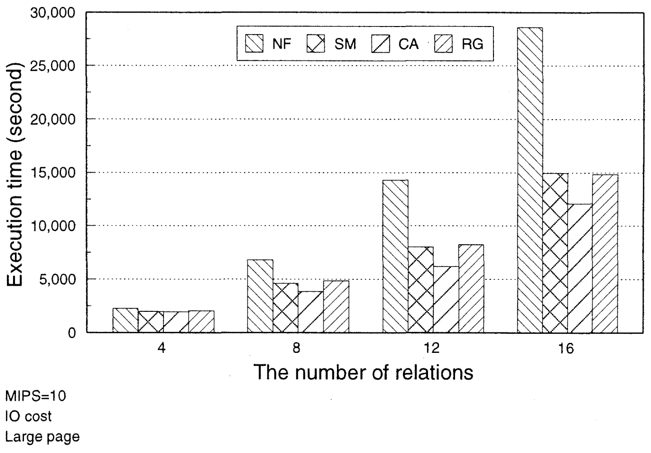


**Fig. 10.** The I/O cost for each scheme

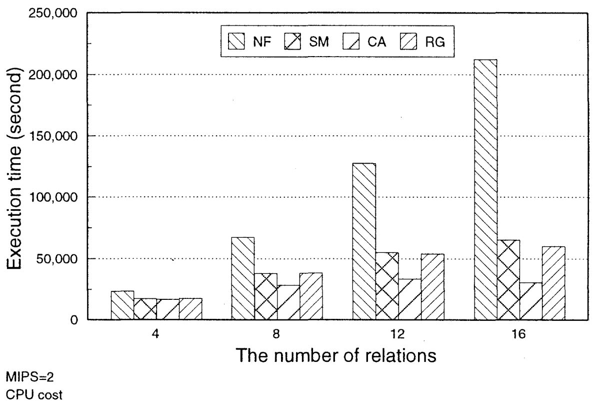


**Fig. 11.** The CPU cost for each scheme when MIPS=2

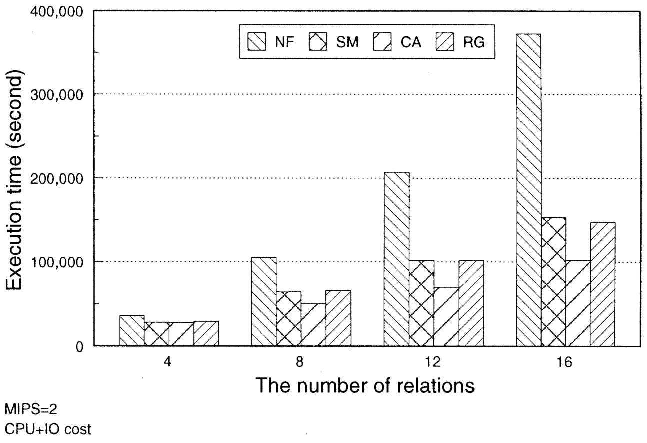


**Fig. 12.** The total cost for each scheme when MIPS=2

### 4.3.2 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 Fig. 11. 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 Fig. 8. It can be seen from Figs. 8 and 11 that queries in Experiment 2 are CPU bound under NF. Figures 7 and 11 show

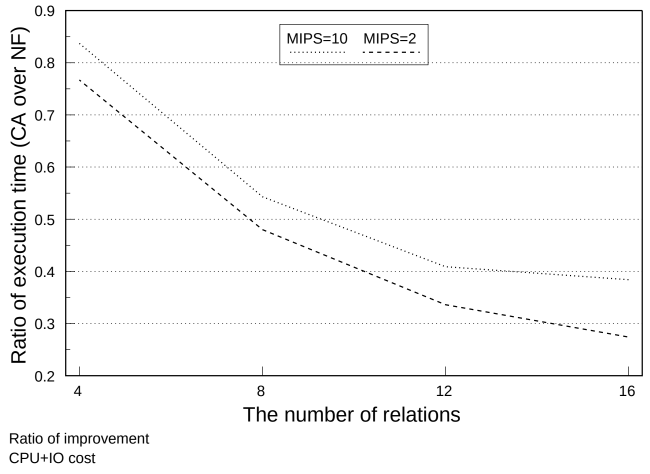


**Fig. 13.** Execution cost ratio of CA to NF

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

| | Standard dev | Maximum | Minimum |
|---|---|---|---|
| NF | 15632 | 293295 | 184366 |
| SM | 20028 | 180580 | 67206 |
| CA | 21485 | 165182 | 41077 |
| RG | 27389 | 206659 | 60207 |

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. Fig. 12 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 HFs 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 Fig. 13, 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 Fig. 13 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 13 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 HF 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.

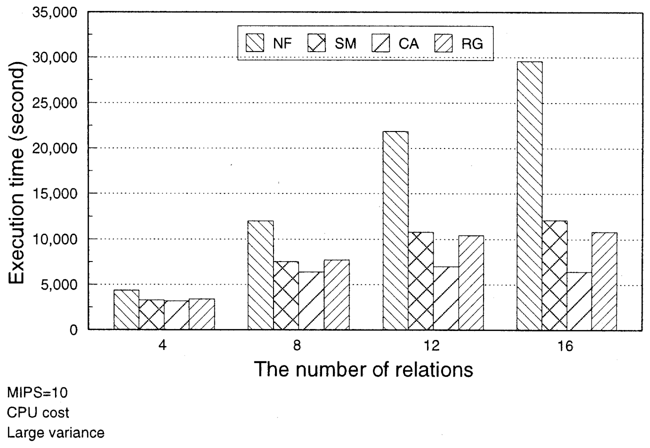


**Fig. 14.** The CPU cost for each scheme when MIPS=10

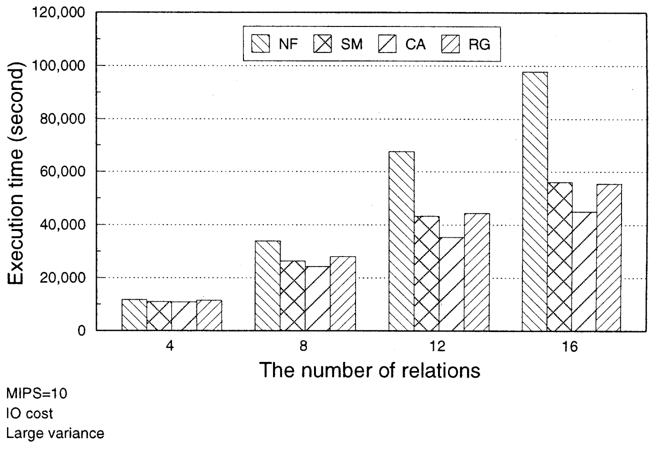
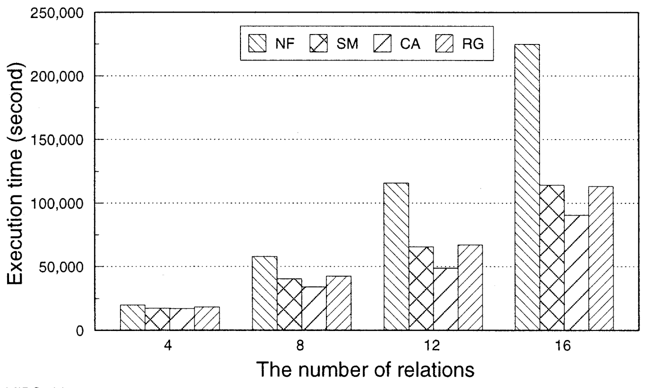


**Fig. 15.** The I/O cost for each scheme when MIPS=10

### 4.3.3 Experiment 3: 10 MIPS CPU with $attv$ = 400K and $carv$ = 600K

In Experiment 3, the CPU speed was set to 10 MIPS while $attv$ and $carv$ were changed to 400K and 600K, respectively. By changing the variances of relation cardinalities and attribute cardinalities, the effectiveness of HFs on join operations with varied relation and attribute cardinalities can be studied. Figures 14, 15, and 16 show, respectively, the CPU cost, the I/O cost, and the total cost for each scheme. Compared to the results in Experiment 1, these three figures indicate that the effectiveness of applying HFs is very stable when the variances of relation cardinalities and attribute cardinalities increase. As before, this experiment shows that CA is the best scheme among all schemes evaluated.

The minimum, maximum, and standard deviation for the four schemes in Experiment 3 with $sn$=12 are given in Table 5, which shows that CA has not only the smallest maximum and minimum execution times, but also the smallest standard deviation, meaning that CA is more stable than NF when the variance of relation cardinalities increases. This is different from what we observed from the results in the previous two experiments. Note that in the presence of a larger variance for relation cardinalities performance of NF changes drastically. On the other hand, performance of the other three schemes, due to the applications of HFs, is not so



MIPS=10
CPU+IO cost
Large variance

**Fig. 16.** The total cost for each scheme when MIPS=10

**Table 5.** Statistics for the cost of each scheme for large relation variance when $sn$=12 and MIPS=10

| | Standard dev | Maximum | Minimum |
|---|---|---|---|
| NF | 35171 | 296913 | 44581 |
| SM | 28548 | 227314 | 23595 |
| CA | 27203 | 211628 | 15715 |
| RG | 59418 | 246145 | 19439 |

sensitive to a variance change as NF, explaining the reason that CA has the smallest standard deviation in this experiment. In addition, RG continues to have the largest standard deviation and is outperformed by CA and SM.

## 5 Conclusions

In this paper, we explored an approach of interleaving a bushy execution tree with HFs to improve the execution of multi-join queries. An efficient scheme to determine an effective sequence of HFs for a bushy execution tree has been developed, where the HFs 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 HFs 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 insight into the use of HFs. It is experimentally shown that the application of HFs 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.

Although simulation was conducted assuming sort-merge joins were employed, the proposed scheme is applicable to other join methods. Note that in dealing with pipelined hash joins, HFs can be applied within a pipeline segment to reduce the sizes of hash tables and improve the pipeline execution. In the case of handling a segmented right-deep tree [5], which is a bushy tree with right-deep subtrees, one can use segment sequence numbers instead of join sequence numbers, to properly insert HFs into the bushy tree among different segments. This is a matter for future research.

## Appendix

### *A. Expected resulting cardinalities of joins*

**Proposition.** *Let $G = (V, E)$ be a join query graph. $G_B = (V_B, E_B)$ is a connected subgraph of G. Let $R_1$, $R_2$, ..., $R_q$ be the relations corresponding to vertices in $V_B$, $A_1$, $A_2$, ..., $A_r$ be the distinct attributes associated with edges in $E_B$ and $m_i$ be the number of different vertices (relations) that edges with attribute $A_i$ are incident to. Suppose $R_M$ is the relation resulting from all the join operations between relations in $G_B$ and $N_T(G_B)$ is the expected number of tuples in $R_M$. Then,*

$$N_T(G_B) = \frac{\Pi_{i=1}^{q}|R_i|}{\Pi_{i=1}^{r}|A_i|^{m_i-1}}.$$

## References

1. Babb E (1979) Implementing a relational database by means of specialized hardware. ACM Trans Database Syst, 4(1):1–29
2. Bernstein PA, Chiu D-MW (1981) Using semi-joins to solve relational queries. J ACM 28(1):25–40
3. Bitton D, Gray J (1988) Disk shadowing. Proc the 14th International Conference on Very Large Data Bases
4. Boral H, Alexander W, et al (1990) Prototyping Bubba, a highly parallel database system. IEEE Trans Knowl Data Eng, 2(1):4–24
5. Chen M-S, Lo M-L, Yu PS, Young HC (1995) Applying segmented right-deep trees to pipelining multiple hash joins. IEEE Trans Knowl Data Eng, 7(4):656–668
6. Chen M-S, Yu PS (1992) Interleaving a join sequence with semijoins in distributed query processing. IEEE Trans Parallel Distrib Syst, 3(5):611–621
7. Chen M-S, Yu PS, Wu K-L (1995) Optimization of Parallel Execution for Multi-Join Queries. IEEE Trans Knowl Data Eng, 8(3): 416–428
8. DeWitt DJ, Ghandeharizadeh S, Schneider DA, Bricker A, Hsiao HI, Rasmussen R (1990) The Gamma database machine project. IEEE Trans Knowl Data Eng, 2(1):44–62
9. DeWitt DJ, Gray J (1992) Parallel database systems: the future of high performance database systems. Commun ACM 35(6):85–98
10. Gardy D, Puech C (1989) On the effect of join operations on relation sizes. ACM Trans Database Syst 14(4):574–603
11. Hong W (1992) Exploiting Inter-Operator Parallelism in XPRS. Proc ACM SIGMOD, June, pp 19–28
12. Hong W, Stonebraker M (1991) Optimization of parallel query execution plans in XPRS. Proc 1st Conf Parallel and Distributed Information Systems, December, pp 218–225
13. Hsiao H-I, DeWitt D (1991) A performance study of three high availability data replication strategies. Proc Conference Parallel and Distributed Information Systems, December, pp 79–84
14. Ioannidis YE, Kang YC (1991) Left-deep vs. bushy trees: an analysis of strategy spaces and its implication for query optimization. Proc ACM SIGMOD, May, pp 168–177
15. Jarke M, Koch J (1982) Query optimization in database systems. ACM Computing Surveys, 16(2):111–152
16. Kitsuregawa M, Tanaka H, Moto-Oka T (1984) Architecture and performance of relational algebra machine GRACE. Proc Int Conf Parallel Processing, August, pp 241–250
17. Krishnamurthy R, Boral H, Zaniolo C (1986) Optimization of Nonrecursive Queries. Proc 12th Int Conf Very Large Data Bases, August, pp 128–137
18. Lo M-L, Chen M-S, Ravishankar CV, Yu PS (1993) On optimal processor allocation to support pipelined hash joins. Proc ACM SIGMOD, May, pp 69–78
19. Lorie RA, Daudenarde J-J, Stamos JW, Young HC (1991) Exploiting database parallelism in a message-passing multiprocessor. IBM J Res Dev 35(5/6):681–695

20. Lu H, Shan M-C, Tan K-L (1991) Optimization of multi-way join queries for parallel execution. Proc 17th Int Conf Very Large Data Bases, September, pp 549–560
21. Mishra P, Eich MH (1992) Join processing in relational databases. ACM Computing Surveys 24(1):63–113
22. Pirahesh H, Mohan C, Cheng J, Liu TS, Selinger P (1990) Parallelism in relational data base systems: architectural issues and design approaches. Proc 2nd Int Sympos Databases in Parallel and Distributed Systems, July, pp 4–29
23. Roussopoulos N, Kang H (1991) A pipeline N-way join algorithm based on the 2-way semijoin program. IEEE Trans Knowl Data Eng, 3(4):461–473
24. Schneider D (1990) Complex query processing in multiprocessor database machines. Tech Rep 965, Computer Science Department, University of Wisconsin, Madison
25. Schneider D, DeWitt DJ (1989) A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. Proc ACM SIGMOD, pp 110–121
26. Selinger PG, Astrahan MM, Chamberlin DD, Lorie RA, Price TG (1979) Access path selection in a relational database management system. Proc ACM SIGMOD, pp 23–34
27. Stonebraker M, Katz R, Patterson D, Ousterhout J (1988) The design of XPRS. Proc 14th Int Conf Very Large Data Bases, pp 318–330
28. Swami A (1989) Optimization of large join queries: combining heuristics with combinatorial techniques. Proc ACM SIGMOD, pp 367–376
29. Swami A, Gupta A (1988) Optimization of large join queries. Proc ACM SIGMOD, pp 8–17
30. Teradata (1985) DBC/1012 Database computer system manual release 2.0. Tech Rep Doc C10-0001-02, Teradata Corporation
31. Valduriez P, Gardarin G (1984) Join and semijoin algorithms for a multiprocessor database machine. ACM Trans Database Syst 9(1):133–161
32. Walton CB, Dale AG, Jenevein RM (1991) A taxonomy and performance model of data skew effects in parallel joins. Proc 17th Int Conf Very Large Data Bases, September, pp 537–548
33. Yao SB (1977) Approximating block access in database organizations. Commun ACM 20:260–261
34. Yu PS, Chen M-S, Heiss H, Lee SH (1992) On workload characterization of relational database environments. IEEE Trans Software Eng 18(4):347–355