Join Processing in Database Systems with Large Main Memories

LEONARD D. SHAPIRO
North Dakota State University

We study algorithms for computing the equijoin of two relations in a system with a standard architecture but with large amounts of main memory. Our algorithms are especially efficient when the main memory available is a significant fraction of the size of one of the relations to be joined; but they can be applied whenever there is memory equal to approximately the square root of the size of one relation. We present a new algorithm which is a hybrid of two hash-based algorithms and which dominates the other algorithms we present, including sort-merge. Even in a virtual memory environment, the hybrid algorithm dominates all the others we study.

Finally, we describe how three popular tools to increase the efficiency of joins, namely filters, Babb arrays, and semijoins, can be grafted onto any of our algorithms.

Categories and Subject Descriptors: H.2.0 [**Database Management**]: General; H.2.4 [**Database Management**]: Systems—*query processing*; H.2.6 [**Database Management**]: Database Machines

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Hash join, join processing, large main memory, sort-merge join

1. INTRODUCTION

Database systems are gaining in popularity owing to features such as data independence, high-level interfaces, concurrency control, crash recovery, and so on. However, the greatest drawback to database management systems (other than cost) is the inefficiency of full-function database systems, compared to customized programs; and one of the most costly operations in database processing is the join. Traditionally the most effective algorithm for executing a join (if there are no indices) has been sort-merge [4]. In [6] it was suggested that the existence of increasingly inexpensive main memory makes it possible to use hashing techniques to execute joins more efficiently than sort-merge. Here we extend these results.

Some of the first research on joins using hashing [14, 21] concerned multiprocessor architectures. Our model assumes a "vanilla" computer architecture, that is, a uniprocessor system available in the market today. Although the lack of parallel processing in such systems deprives us of much of the potential speed of

Author's address: Department of Computer Science, North Dakota State University, Fargo, ND 58105.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1986 ACM 0362-5915/86/0900-0239 $00.75

join processing on multiprocessor systems, our algorithms can be implemented on current systems, and they avoid the complex synchronization problems of some of the more sophisticated multiprocessor algorithms.

Our algorithms require significant amounts of main memory to execute most efficiently. We assume it is not unreasonable to expect that the database system can assign several megabytes of buffer space to executing a join. (Current VAX systems can support 32 megabytes of real memory with 64K chips [8]; and it is argued in [10] that a system can be built with existing technology that will support tens of gigabytes of main memory, and which appears to a programmer to have a standard architecture.)

We will see that our algorithms are most effective when the amount of real memory available to the process is close to the size of one of the relations.

The word "large" in the title refers to a memory size large enough that it is not uncommon for all, or a significant fraction, of one of the relations to be joined to fit in main memory. This is because the minimum amount of memory required to implement our algorithms is approximately the square root of the size of one of the relations (measured in physical blocks). This allows us to process rather large relations in main memories which by today's standards might not be called large. For example, using our system parameters and 4 megabytes of real memory as buffer space, we can join two relations, using our most efficient algorithm, if the smaller of the two relations is at most 325 megabytes.

We show that with sufficient main memory, and for sufficiently large relations, the most efficient algorithms are hash-based. We present two classes of hash-based algorithms, one (simple hash) that is most efficient when most of one relation fits in main memory and another (GRACE) that is most efficient when much less of the smaller relation fits. We then describe a new algorithm, which is a hybrid of simple and GRACE, that is the most efficient of those we study, even in a virtual memory environment. This is in contrast to current commercial database systems, which find sort-merge-join to be most efficient in many situations and do not implement hash joins.

In Section 2 we present four algorithms for computing an equijoin, along with their cost formulas. The first algorithm is sort-merge, modified to take advantage of large main memory. The next is a very simple use of hashing, and another is based on GRACE, the Japanese fifth-generation project's database machine [14]. The last is a hybrid of the simple and GRACE algorithms. We show in Section 3 that for sufficiently large relations the hybrid algorithm is the most efficient, inclusive of sort-merge, and we present some analytic modeling results. All our hashing algorithms are based on the idea of partitioning: we partition each relation into subsets which can (on the average) fit into main memory. In Section 3 we assume that all the partitions can fit into main memory, and in Section 4 discuss how to deal with the problem of partition overflow. In Section 5 we describe the effect of using virtual memory in place of some main memory. In Section 6 we discuss how to include in our algorithms other tools that have become popular in database systems, namely selection filters, semijoin strategies, and Babb arrays.

A description of these algorithms, similar to that in Section 2, and analytic modeling results similar to those in the second half of Section 3, appeared in [6].

In [5] it is shown that hashing is preferable to nested-loop and sort-merge algorithms for a variety of relational algebra operations—results consistent with those we present. In [7] the results of [6] are extended to the multiprocessor environment, and experimental results are reported. These results support the analyses in [6] and in the present paper, and show that, in the cases reported, if a bit-filtering technique is used (see Section 6) the timings of all algorithms are similar above a certain memory size. A related algorithm, the nested-hash algorithm, is also studied there, and is shown to have performance comparable to the hybrid algorithm for large memory sizes, but to be inferior to hybrid for smaller memory sizes. In [22], the GRACE hash algorithm is studied in depth, including an analysis of the case when more than two phases of processing are needed and an analysis of various partitioning schemes. I/O accesses and CPU time are analyzed separately, and it is shown that GRACE hash-join is superior to merge-join.

1.1 Notation and Assumptions

Our goal is to compute the equijoin of two relations labeled **R** and **S**. We use **M** to denote main memory. We do not count initial reads of **R** or **S** or final writes of the join output because these costs are identical for all algorithms. After the initial reads of **R** and **S**, those relations are not referenced again by our algorithms. Therefore, all I/O in the join processing is of temporary relations. We choose to block these temporary relations at one track per physical block, so each I/O which we count will be of an entire track. Therefore, throughout this paper the term "block" refers to a full track of data. Of course, **R** and **S** may be stored with a different blocking factor. In all the cost formulas and analytic modeling in this paper we use the labels given in Figure 1.

In our model we do not distinguish between sequential and random I/O. This is justified because all reads or writes of any temporary file in our algorithms will be sequential from or to that file. If only one file is active, I/O is sequential in the traditional sense, except that because of our full-track blocking an I/O operation is more likely to cause head movement. If more than one file is active there will be more head movement, but the cost of that extra head movement is assumed negligible. This is why we choose a full track as our blocking factor for the temporary relations.

We assume that **S** is the larger relation, that is, $|\mathbf{R}| \leq |\mathbf{S}|$. We use the "fudge factor", **F**, in Figure 1 to calculate values that are small increments of other values. For example, a hash table for **R** is assumed to occupy $|\mathbf{R}| * \mathbf{F}$ blocks.

In our cost formulas we assume that all selections and projections of **R** and **S** have already been done and that neither relation **R** nor **S** is ordered or indexed. We assume no overlap between CPU and I/O processing. We assume each tuple from **S** joins with at most one block of tuples from **R**. If it is expected that there will be few tuples in the resulting join, it may be appropriate to process only tuple IDs instead of projected tuples, and then at the end translate the TIDs that are output into actual tuple values. We view this as a separate process from the actual join; our formulas do not include this final step.

For the first four sections of this paper we assume that the memory manager allocates a fixed amount of real memory to each join process. The process knows

comp	time to compare keys in main memory
hash	time to hash a key that is in main memory
move	time to move a tuple in memory
swap	time to swap two tuples in memory
IO	time to read or write a block between disk and main memory
F	incremental factor (see below)
$\|\mathbf{R}\|$	number of blocks in **R** relation (similar for **S** and **M**)
$\{\mathbf{R}\}$	number of tuples in **R** (similar for **S**)
$\{M\}_R$	number of **R** tuples that can fit in **M** (similar for **S**)

Fig. 1. Notation used in cost formulas in this paper.

how much memory is allocated to it, and can use this information in designing a strategy for the join. The amount of real memory allocated is fixed throughout the lifetime of the process. In Section 5 we discuss an alternate to this simple memory management strategy.

2. THE JOIN ALGORITHMS

In this section we present four algorithms for computing the equijoin of relations **R** and **S**. One is a modified sort-merge algorithm and the other three are based on hashing.

Each of the algorithms we describe executes in two phases. In phase 1, the relations **R** and **S** are restructured into runs (for sort-merge) or subsets of a partition (for the three hashing algorithms). In phase 2, the restructured relations are used to compute the join.

2.1 Sort-Merge-Join Algorithm

The standard sort-merge-join algorithm [4] begins by producing sorted runs of tuples of **S**. The runs are on the average (over all inputs) twice as long as the number of tuples that can fit into a priority queue in memory [15, p. 254]. This requires one pass over **S**. In subsequent phases, the runs are sorted using an n-way merge. **R** is sorted similarly. After **R** and **S** are sorted they are merged together and tuples with matching join attributes are output. For a fixed relation size, the CPU time to do a sort with n-way merges is independent of n, but I/O time increases as n decreases and the number of phases increases. One should therefore choose the merging factor n to be as large as possible so that the process will involve as few phases as possible. Ideally, only two phases will be needed, one to construct runs and the other to merge and join them. We show that if $|\mathbf{M}|$ is at least $\sqrt{|S|}$, then only two phases are needed to accomplish the join.

Here are the steps of our version of the sort-merge algorithm in the case where there are at least $\sqrt{|S|}$ blocks of memory for the process. (See Figure 2, sort-merge-join).

In the following analysis we use average (over all inputs) values, for instance, $2 * |\mathbf{M}|$ is the length of a run.

(1) Scan **S** and produce output runs using a heap or some other priority queue structure. Do the same for **R**. A run will be $2 * |\mathbf{M}|$ blocks long. Given

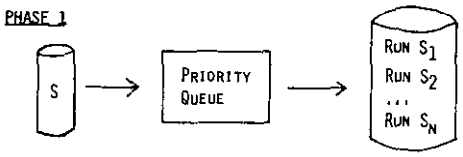
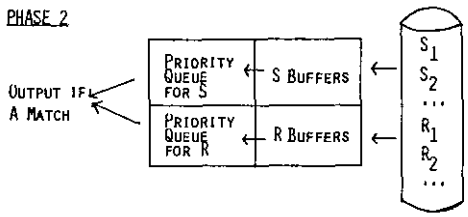


Fig. 2. Sort-merge-join.



that $|\mathbf{M}| \geq \sqrt{|S|}$, the runs will be at least $2\sqrt{|S|}$ blocks in length. Therefore there will be at most

$$\frac{|S|}{2\sqrt{|S|}} = \frac{1}{2}\sqrt{|S|}$$

distinct runs of **S** on the disk. Since **S** is the larger relation, **R** has at most the same number of runs on disk. Therefore there will be at most $\sqrt{|S|}$ runs of **R** and **S** altogether on the disk at the end of phase 1.

(2) Allocate one block of memory for buffer space for each run of **R** and **S**. Merge all the runs of **R** and concurrently merge all the runs of **S**. As tuples of **R** and **S** are generated in sorted order by these merges, they can be checked for a match. When a tuple from **R** matches one from **S**, output the pair.

In step (2), one input buffer is required per run, and there are at most $\sqrt{|S|}$ runs, so if $|\mathbf{M}| \geq \sqrt{|S|}$, there is sufficient room for the input buffers. Extra space is required for merging, but this is negligible since the priority queue contains only one tuple per run.

If the memory manager allocates fewer than $\sqrt{|S|}$ blocks of memory to the join process, more than two phases are needed. We do not investigate this case further; we assume $|\mathbf{M}| \geq \sqrt{|S|}$. If $|\mathbf{M}|$ is greater than $\sqrt{|S|}$, the extra blocks of real memory can be used to store runs between phases, thus saving I/O costs. This is reflected in the last term of the cost formula.

The cost of this algorithm is

$(\{\mathbf{R}\} \log_2\{\mathbf{R}\} + \{\mathbf{S}\} \log_2\{\mathbf{S}\}) * (\text{comp} + \text{swap})$	Manage priority queues in both phases.
$+ (\lvert\mathbf{R}\rvert + \lvert\mathbf{S}\rvert) * \text{IO}$	Write initial runs.
$+ (\lvert\mathbf{R}\rvert + \lvert\mathbf{S}\rvert) * \text{IO}$	Read initial runs.
$+ (\{\mathbf{R}\} + \{\mathbf{S}\}) * \text{comp}$	Join results of final merge.
$- \min(\lvert\mathbf{R}\rvert + \lvert\mathbf{S}\rvert, \lvert\mathbf{M}\rvert - \sqrt{\lvert S\rvert}) * 2 * \text{IO}$	I/O savings if extra memory is available.

2.2 Hashing Algorithms

The simplest use of hashing as a join strategy is the following algorithm, which we call *classic hashing*: build a hash table, in memory, of tuples from the smaller relation **R**, hashed on the joining attribute(s). Then scan the other relation **S** sequentially. For each tuple in **S**, use the hash value for that tuple to probe the hash table of **R** for tuples with matching key values. If a match is found, output the pair, and if not then drop the tuple from **S** and continue scanning **S**.

This algorithm works best when the hash table for **R** can fit into real memory. When most of a hash table for **R** cannot fit in real memory, this classic algorithm can still be used in virtual memory, but it behaves poorly, since many tuples cause page faults. The three hashing algorithms we describe here each extend the classic hashing approach in some way so as to take into account the possibility that a hash table for **R** will not fit into main memory.

If a hash table for the smaller relation **R** cannot fit into memory, each of the hashing algorithms described in this paper calculates the join by partitioning **R** and **S** into disjoint subsets and then joining corresponding subsets. The size of the subsets varies for different algorithms. For this method to work, one must choose a partitioning of **R** and **S** so that computing the join can be done by just joining corresponding subsets of the two relations. The first mention of this method is in [11], and it also appears in [3]. Our use of it is closely related to the description in [14].

The method of partitioning is first to choose a hash function h, and a partition of the values of h into, say, $H_1, \ldots, H_n$. (For example, the negative and nonnegative values of h constitute a partition of h values into two subsets, H_1 and H_2.) Then, one partitions **R** into corresponding subsets $R_1, \ldots, R_n$, where a tuple r of **R** is in R_i whenever $h(r)$ is in H_i. Here by $h(r)$ we mean the hash function applied to the joining attribute of r. Similarly, one partitions **S** into corresponding subsets $S_1, \ldots, S_n$ with s in S_i when $h(s)$ is in H_i. The subsets R_i and S_i are actually buckets, but we refer to them as subsets of the partition, since they are not used as ordinary hash buckets.

If a tuple r in **R** is in R_i and it joins with a tuple s from **S**, then the joining attributes of r and s must be equal, thus $h(r) = h(s)$ and s is in S_i. This is why, to join **R** and **S**, it suffices to join the subsets R_i and S_i for each i.

In all the hashing algorithms we describe we are required to choose a partitioning into subsets of a specified size (e.g., such that **R** is partitioned into two subsets of equal size). Partitioning into specified size subsets is not easy to accomplish if the distribution of the joining attribute of **R** is not well understood. In this section we describe the hashing algorithms as if bucket overflow never occurs, then in Section 4 we describe how to deal with the problem.

Each of the three algorithms we describe uses partitioning, as described above. Each proceeds in two phases: the first is to partition each relation. The second phase is to build hash table(s) for **R** and probe for matches with each tuple of **S**. The first algorithm we describe, simple hashing, does as little of the first phase—partitioning—as possible on each step, and goes right into building a hash table and probing. It performs well when most of **R** can fit in memory. The next algorithm, GRACE hash, does all of the first phase at once, then turns to the second phase of building hash tables and probing. It performs relatively well

when little of **R** can fit in memory. The third algorithm, hybrid hash, combines the two, doing all partitioning on the first pass over each relation and using whatever memory is left to build a hash table. It performs well over a wide range of memory sizes.

2.3 Simple Hash-Join Algorithm

If a hash table containing all of **R** fits into memory (i.e., if $|R| * F \leq |M|$), the simple hash-join algorithm which we define here is identical to what we have called classic hash-join. If there is not enough memory available, our simple hash-join scans **R** repeatedly, each time partitioning off as much of **R** as can fit in a hash table in memory. After each scan of **R**, **S** is scanned and, for tuples corresponding to those in memory, a probe is made for a match (see Figure 3, simple hash-join).

The steps of our simple hash-join algorithm are

(1) Let $\mathbf{P} = \min(|\mathbf{M}|, |\mathbf{R}| * \mathbf{F})$. Choose a hash function h and a set of hash values so that **P**/**F** blocks of **R** tuples will hash into that set. Scan the (smaller) relation **R** and consider each tuple. If the tuple hashes into the chosen range, insert the tuple into a **P**-block hash table in memory. Otherwise, pass over the tuple and write it into a new file on disk.

(2) Scan the larger relation **S** and consider each tuple. If the tuple hashes into the chosen range, check the hash table of **R** tuples in memory for a match and output the pair if a match occurs. Otherwise, pass over the tuple and write it to disk. Note that if key values of the two relations are distributed similarly, there will be $P/F * |S|/|R|$ blocks of the larger relation **S** processed in this pass.

(3) Repeat steps (1) and (2), replacing each of the relations **R** and **S** by the set of tuples from **R** and **S** that were "passed over" and written to disk in the previous pass. The algorithm ends when no tuples from **R** are passed over.

This algorithm performs particularly well when most of **R** fits into main memory. In that case, most of **R** (and **S**) are touched only once, and only what cannot fit into memory is written out to disk and read in again. On the other hand, when there is little main memory this algorithm behaves poorly, since in that case there are many passes and both **R** and **S** are scanned over and over again. In fact, this algorithm operates as specified for any amount of memory, but to be consistent with the other hash-based algorithms we assume it is undefined for less than $\sqrt{F|R|}$ blocks of memory.

We assume here and elsewhere that the same hash function is used both for partitioning and for construction of the hash tables.

In the following formula and in later formulas we must estimate the number of compares required when the hash table is probed for a match. This amounts to estimating the number of collisions. We have chosen to use the term comp * **F** for the number of compares required. Although this term is too simple to be valid in general, when **F** is 1.4 (which is the value we use in our analytic modeling) it means that for a hash table with a load factor of 71 percent the estimated number of probes is 1.4. This is consistent with the simulations reported in [18].

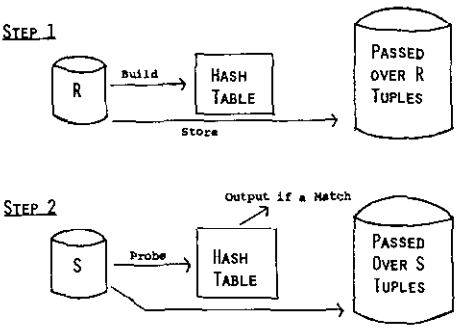


Fig. 3. Simple hash-join.

The algorithm requires

$$\left\lceil \frac{|R| * F}{|M|} \right\rceil$$

passes to execute, where $\lceil\ \rceil$ denotes the ceiling function. We denote this quantity by **A**. Note that on the ith pass, $i = 1, \ldots, \mathbf{A} - 1$, there are

$$\{R\} - i * \frac{\{M\}_R}{F}$$

tuples of **R** passed over.

The cost of the algorithm is

$$+ \left[A * \{R\} - \frac{A * (A-1)}{2} * \frac{\{M\}_R}{F} \right] * (\text{hash} + \text{move}).$$

Hash and move **R** and passed-over tuples in R.

$$+ \left[A * \{S\} - \frac{A * (A-1)}{2} * \frac{\{M\}_S}{F} \right] * (\text{hash} + \text{move}).$$

Hash and move **S** and passed-over tuples in **S**.

$- \{S\} * \text{move}$ — On each pass, passed-over tuples of **S** are moved into the buffer and others result in probing the hash table for a match. The latter are not moved, yet they are counted as being moved in the previous term. This adjustment corrects that.

$+ \{\mathbf{S}\} * \text{comp} * \mathbf{F}$ — Check each tuple of **S** for a match.

$$+ \left[(A-1) * |R| - \frac{A * (A-1)}{2} * \frac{|M|}{F} \right] * 2 * \text{IO}.$$

Write and read passed-over tuples in **R**.

$$+ \left[(A-1) * |S| - \frac{A * (A-1)}{2} * \frac{|M|}{F} * \frac{|S|}{|R|} \right] * 2 * \text{IO}.$$

Write and read passed-over tuples in **S**.

2.4 GRACE Hash-Join Algorithm

As outlined in [14], the GRACE hash-join algorithm executes as two phases. The first phase begins by partitioning **R** and **S** into corresponding subsets, such that **R** is partitioned into sets of approximately equal size. During the second phase of the GRACE algorithm the join is performed using a hardware sorter to execute a sort-merge algorithm on each pair of sets in the partition.

Our version of the GRACE algorithm differs from that of [14] in two ways. First, we do joining in the second phase by hashing, instead of using hardware sorters. Second, we use only $\sqrt{F|R|}$ blocks of memory for both phases; the rest is used to store as much of the partitions as possible so they need not be written to disk and read back again. The algorithm proceeds as follows, assuming there are $\sqrt{F|R|}$ blocks of memory (see Figure 4):

(1) Choose a hash function h, and a partition of its hash values, so that **R** will be partitioned into $\sqrt{F|R|}$ subsets of approximately equal size.[1] Allocate $\sqrt{F|R|}$ blocks of memory, each to be an output buffer for one subset of the partition of **R**.
(2) Scan **R**. Using h, hash each tuple and place it in the appropriate output buffer. When an output buffer fills, it is written to disk. After **R** has been completely scanned, flush all output buffers to disk.
(3) Scan **S**. Using h, the same function used to partition **R**, hash each tuple and place in the appropriate output buffer. When an output buffer fills, it is written to disk. After **S** has been completely scanned, flush all output buffers to disk.

Steps (4) and (5) below are repeated for each set R_i, $1 \le i \le \sqrt{F|R|}$, in the partition for **R**, and its corresponding set S_i.

(4) Read R_i into memory and build a hash table for it.
We pause to check that a hash table for R_i can fit in memory. Assuming that all the sets R_i are of equal size, since there are $\sqrt{F|R|}$ of them, each of the sets R_i will be

$$\frac{|R|}{\sqrt{F|R|}} = \sqrt{\frac{|R|}{F}}$$

blocks in length. A hash table for each subset R_i will therefore require

$$F\sqrt{\frac{|R|}{F}} = \sqrt{F|R|}$$

blocks of memory, and we have assumed at least this much real memory.
(5) Hash each tuple of S_i with the same hash function used to build the hash table in (4). Probe for a match. If there is one, output the result tuple, otherwise proceed with the next tuple of S_i.

What if there are more or less than $\sqrt{F|R|}$ blocks of memory available? Just as with sort-merge-join, we do not consider the case when less than this minimum

[1] Our assumption, that a tuple of **S** joins with at most one block of tuples of **R**, is used here. If **R** contains many tuples with the same joining attribute value, then this partitioning may not be possible.

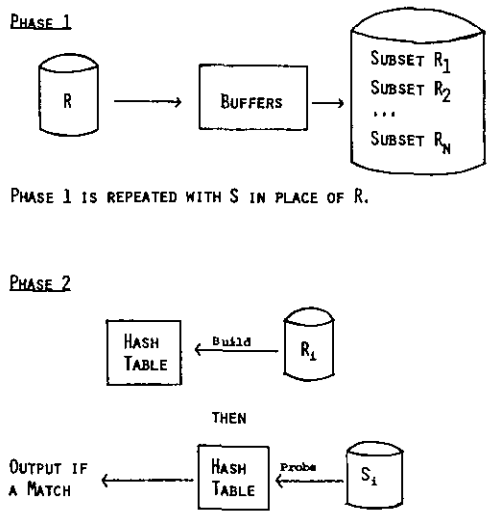


Fig. 4. GRACE hash-join.

number of blocks is available, and if more blocks are available, we use them to store subsets of **R** and/or **S** so they need not be written to and read from disk.

This algorithm works very well when there is little memory available, because it avoids repeatedly scanning **R** and **S**, as is done in simple hash. Yet when most of **R** fits into memory, GRACE join does poorly since it scans both **R** and **S** twice.

One advantage of using hash in the second phase, instead of sort-merge, as is done by the designers of the GRACE machine, is that subsets of **S** can be of arbitrary size. Only **R** needs to be partitioned into subsets of approximately equal size. Since partition overflow can cause significant problems (see Section 4), this is an important advantage.

The cost of this algorithm is

$(\{\mathbf{R}\} + \{\mathbf{S}\} * (\text{hash} + \text{move})$	Hash tuple and move to output buffer.
$+ (\lvert\mathbf{R}\rvert + \lvert\mathbf{S}\rvert) * \text{IO}$	Write partitioned relations to disk.
$+ (\lvert\mathbf{R}\rvert + \lvert\mathbf{S}\rvert) * \text{IO}$	Read partitioned sets.
$+ \{\mathbf{R}\} * (\text{hash} + \text{move})$	Build hash tables in memory.
$+ \{\mathbf{S}\} * (\text{hash} + \text{comp} * \mathbf{F})$	Probe for a match.
$- \min(\lvert\mathbf{R}\rvert + \lvert\mathbf{S}\rvert, \lvert\mathbf{M}\rvert \sqrt{F \lvert R \rvert}) * 2 * \text{IO}$	IO savings if extra memory is available.

2.5 Hybrid Hash-Join Algorithm

Hybrid hash combines the features of the two preceding algorithms, doing both partitioning and hashing on the first pass over both relations. On the first pass, instead of using memory as a buffer as is done in the GRACE algorithm, only as many blocks (**B**, defined below) as are necessary to partition **R** into sets that can fit in memory are used. The rest of memory is used for a hash table that is processed at the same time that **R** and **S** are being partitioned (see Figure 5).

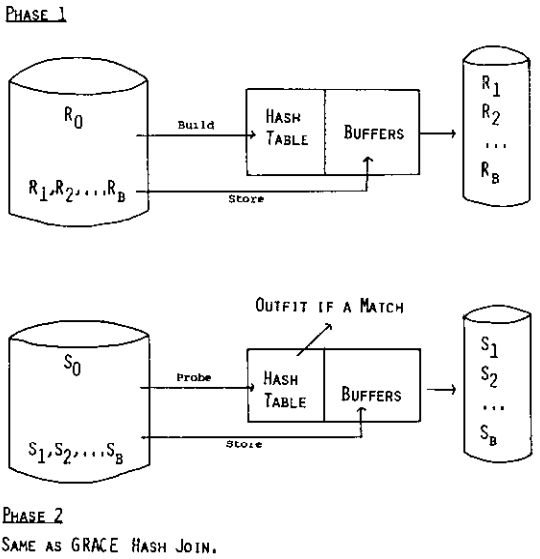


Fig. 5. Hybrid hash-join.

Here are the steps of the hybrid hash algorithm. If $|R| * F \leq M$, then a hash table for **R** will fit in real memory, and hybrid hash is identical to simple hash in this case.

(1) Let

$$\mathbf{B} = \left\lceil \frac{|R| * F - |M|}{|M| - 1} \right\rceil.$$

There will be $\mathbf{B} + 1$ steps in the hybrid hash algorithm. (To motivate the formula for **B**, we note that it is approximately equal to the number of steps in simple hash. The small difference is due to setting aside some real memory in phase 1 for a hash table for R_0.) First, choose a hash function h and a partition of its hash values which will partition **R** into $R_0, \ldots, R_B$, such that a hash table for R_0 has $|\mathbf{M}| - \mathbf{B}$ blocks, and $R_1, \ldots, R_B$ are of equal size. Then allocate **B** blocks in memory to **B** output buffers. Assign the other $|\mathbf{M}| - \mathbf{B}$ blocks of memory to a hash table for R_0.

(2) Assign the ith output buffer block to R_i for $i = 1, \ldots, \mathbf{B}$. Scan **R**. Hash each tuple with h. If it belongs to R_0 it will be placed in memory in a hash table. Otherwise it belongs to R_i for some $i > 0$, so move it to the ith output buffer block. When this step has finished, we have a hash table for R_0 in memory, and $R_1, \ldots, R_B$ are on disk.

(3) The partition of **R** corresponds to a partition of **S** compatible with h, into sets $S_0, \ldots, S_B$. Assign the ith output buffer block to S_i for $i = 1, \ldots, \mathbf{B}$. Scan **S**, hashing each tuple with h. If the tuple is in S_0, probe the hash table in memory for a match. If there is a match, output the result tuple, otherwise drop the tuple. If the tuple is not in S_0, it belongs to S_i for some $i > 0$, so move it to the ith output buffer block. Now $R_1, \ldots, R_B$ and $S_1, \ldots, S_B$ are on disk.

Repeat steps (4) and (5) for $i = 1, \ldots, \mathbf{B}$.

(4) Read R_i and build a hash table for it in memory.

(5) Scan S_i, hashing each tuple, and probing the hash table for R_i, which is in memory. If there is a match, output the result tuple, otherwise toss the **S** tuple.

We omit the computation that shows that a hash table for R_i will actually fit in memory. It is similar to the above computation for the GRACE join algorithm.

For the cost computation, denote by q the quotient $|R_0|/|R|$, namely the fraction of **R** represented by R_0. To calculate the cost of this join we need to know the size of S_0, and we estimate it to be $q * |\mathbf{S}|$. Then the fraction of **R** and **S** sets remaining on the disk after step (3) is $1 - q$. The cost of the hybrid hash join is

$(\{\mathbf{R}\} + \{\mathbf{S}\}) * \text{hash}$	Partition **R** and **S**.
$+ (\{\mathbf{R}\} + \{\mathbf{S}\}) * (1 - q) * \text{move}$	Move tuples to output buffers.
$+ (\lvert\mathbf{R}\rvert + \lvert\mathbf{S}\rvert) * (1 - q) * \text{IO}$	Write from output buffers.
$+ (\lvert\mathbf{R}\rvert + \lvert\mathbf{S}\rvert) * (1 - q) * \text{IO}$	Read subsets into memory.
$+ (\{\mathbf{R}\} + \{\mathbf{S}\}) * (1 - q) * \text{hash}$	Build hash tables for **R** and hash to probe for **S** during (4) and (5).
$+ \{\mathbf{R}\} * \text{move}$	Move tuples to hash tables for **R**.
$+ \{\mathbf{S}\} * \text{comp} * F$	Probe for each tuple of **S**.

At the cost of some complexity, each of the above algorithms could be improved by not flushing buffers at the end of phase 1. The effect of this change is analyzed in Section 5.

3. COMPARISON OF THE FOUR JOIN ALGORITHMS

We begin by showing that when **R** and **S** are sufficiently large the hybrid algorithm dominates the other two hash-based join algorithms, then we show that hybrid also dominates sort-merge for sufficiently large relations. In fact, we also show that GRACE dominates sort-merge, except in some cases where **R** and **S** are close in size. Finally, we present results of analytic modeling of the four algorithms. Our assumption that **R** (and therefore **S**) are sufficiently large, along with our previous assumption that $|\mathbf{M}|$ is at least $\sqrt{|S|}$ (sort-merge) of $\sqrt{F|R|}$ (hash-based algorithms) means that we can also assume $|\mathbf{M}|$ to be large. The precise definition of "large" depends on system parameters, but it will typically suffice that $\{\mathbf{R}\}$ be at least 1000 and $|\mathbf{M}|$ at least 5.

First we indicate why hybrid dominates simple hash-join. We assume less than half of a hash table for **R** fits in memory because otherwise the hybrid and simple join algorithms are identical. Denoting $|\mathbf{R}| * \mathbf{F}/2 - |\mathbf{M}|$ by **E**, our assumption means that $\mathbf{E} > 0$. If we ignore for a moment the space requirements for the output buffers for both simple and hybrid hash, the I/O and CPU costs for both methods are identical, except that some tuples written to disk in the simple hash-join are processed[2] more than once, whereas each is processed only once in hybrid

[2] By *processing* we mean, for tuples of **R**, one hash and one move of each tuple plus one read and one write for each block. For **S** tuples we mean one hash and one move or compare per tuple plus two I/Os per block.

hash-join. In fact, $|\mathbf{R}| - 2 * |\mathbf{M}|/F = 2 * \mathbf{E}/F$ blocks of $\mathbf{R}$ will be processed more than once by simple hash (and similarly for some $\mathbf{S}$ tuples). So far hybrid is ahead by the cost of processing at least $2 * \mathbf{E}/F$ blocks of tuples. Now consider the space requirements for output buffers, which we temporarily ignored above. Simple hash uses only one output buffer. Hybrid uses approximately $(|\mathbf{R}| * F/|\mathbf{M}|) - 1$ output buffers,[3] that is, $(|\mathbf{R}| * F/|\mathbf{M}|) - 2$ more than simple hash-join uses, and $|\mathbf{R}| * F/|\mathbf{M}| - 2 = 2\mathbf{E}/|\mathbf{M}|$. Therefore hybrid must process the extra $2 * \mathbf{E}/|\mathbf{M}|$ blocks in the second phase, since space for them is taken up by buffers. In total, hybrid is ahead by the cost of processing $(2 * \mathbf{E}/F) - (2 * \mathbf{E}/|\mathbf{M}|)$ blocks,which is clearly a positive number. We conclude that hybrid dominates simple hash-join.

Next we indicate why hybrid dominates GRACE. If we compare the cost formulas, they are identical in CPU time, except that some terms in the hybrid cost formula are multiplied by $(1 - q)$. Since $q \leq 1$, hybrid dominates GRACE in CPU costs. The two algorithms read and write the following number of blocks:

$$\text{GRACE: } |\mathbf{R}| + |\mathbf{S}| - \min(|\mathbf{R}| + |\mathbf{S}|, |\mathbf{M}| - \sqrt{F\,|R|}).$$
$$\text{Hybrid: } |\mathbf{R}| + |\mathbf{S}| - q * (|\mathbf{R}| + |\mathbf{S}|).$$

To show that I/O costs for GRACE are greater than those for hybrid, it suffices to prove that

$$q * (|\mathbf{R}| + |\mathbf{S}|) > |\mathbf{M}| - \sqrt{F\,|R|}.$$

Since $|\mathbf{S}| > 0$, we can discard it in the preceding formula. Since $q * |\mathbf{R}| = |R_0| = |\mathbf{M}| - \mathbf{B}$, it suffices to prove that

$$\sqrt{F\,|R|} \geq \mathbf{B}.$$

But $\mathbf{B}$ was the least number of buffers necessary to partition $\mathbf{R} - R_0$ into sets which fit in memory. From the description of the GRACE algorithm we know that $\sqrt{F\,|R|}$ buffers are always enough to partition all of $\mathbf{R}$ into sets which can fit in memory, so $\mathbf{B}$ cannot be more than $\sqrt{F\,|R|}$.

We have proved that hybrid dominates both the simple and the GRACE hash-join algorithms. Now we compare the hash-join algorithms with sort-merge.

When a hash table for $\mathbf{R}$ can fit in main memory, it is clear that hybrid hash will outperform sort-merge. This is because when a hash table for $\mathbf{R}$ can fit in real memory there are no I/O costs, and CPU costs are, with slight rearranging:

$$\text{Hybrid: } \begin{cases} \{\mathbf{R}\} * [\text{hash} + \text{move}] + \\ \{\mathbf{S}\} * [\text{hash} + \text{comp} * \mathbf{F}]. \end{cases}$$

$$\text{Sort: } \begin{cases} \{\mathbf{R}\} * [\text{comp} + (\log_2\{\mathbf{R}\}) * (\text{comp} + \text{swap})] + \\ \{\mathbf{S}\} * [\text{comp} + (\log_2\{\mathbf{S}\}) * (\text{comp} + \text{swap})]. \end{cases}$$

Since the times to hash and to compare are similar on any system, and swap is more expensive than move or comp $* \mathbf{F}$, the log terms will force sort-merge to be more costly except when $\mathbf{R}$ is very small.

[3] Here we have used the formula for $\mathbf{B}$ above, and have assumed $|\mathbf{M}|$ is large enough that $|\mathbf{M}| - 1$ can be approximated by $|\mathbf{M}|$.

comp	compare keys	3 microseconds
hash	hash a key	9 microseconds
move	move a tuple	20 microseconds
swap	swap two tuples	60 microseconds
IO	read/write of a block	30 milliseconds
F	incremental factor	1.4
$\|\mathbf{R}\|$	size of **R**	800 blocks
$\|\mathbf{S}\|$	size of **S**	1600 blocks
$\{\mathbf{R}\}/\|\mathbf{R}\|$	number of **R** tuples/block	250
$\{\mathbf{S}\}/\|\mathbf{S}\|$	number of **S** tuples/block	250
	block size	25,000 bytes

Fig. 6. System parameters used in modeling in this paper.

To show that GRACE typically dominates sort-merge, the previous argument can be extended as follows: first we show that GRACE typically has lower I/O costs than sort-merge. The runs generated by sort-merge and the subsets generated by GRACE are of the same size in total, namely $|\mathbf{R}| + |\mathbf{S}|$, so when memory is at the minimum ($\sqrt{|S|}$ for sort-merge and $\sqrt{F\,|R|}$ for GRACE), I/O costs, which consist of writing and reading the runs or subsets, are identical. More real memory results in equal savings, so GRACE has higher I/O costs only if $\sqrt{F\,|R|} > \sqrt{|S|}$, which is atypical since $|\mathbf{R}|$ is the smaller relation.

Next we compare the CPU costs of GRACE and sort-merge. The CPU cost of GRACE is

$$\text{GRACE: } \begin{cases} \{\mathbf{R}\} * (2 * \text{hash} + 2 * \text{move}) + \\ \{\mathbf{S}\} * (2 * \text{hash} + \text{comp} * \mathbf{F} + \text{move}). \end{cases}$$

As with hybrid's CPU time, the coefficients of $\{\mathbf{R}\}$ and $\{\mathbf{S}\}$ are similar except for the logarithm terms, which force sort-merge to be more costly. We conclude that GRACE dominates sort-merge except when **R** is small or when $\sqrt{F\,|R|} > \sqrt{|S|}$.

We have modeled the performance of the four join algorithms by numerically evaluating our formulas for a sample set of system parameters given in Figure 6. We should note that all the modelings of the hash-based algorithms are somewhat optimistic, since we have assumed no partition overflow. We discuss in Section 4 ways to deal with partition overflow.

In Figure 7 we display the relative performance of the four join algorithms as described above. As we have noted, each algorithm requires a minimum amount of main memory. For the relations modeled in Figure 7, the minimum memory size for sort-merge is 1.1 megabytes, and for hash-based algorithms the minimum memory size is 0.8 megabytes.

Among hash algorithms, simple and our modification of GRACE join each perform as expected, with simple doing well for high memory values and GRACE for low memory. Hybrid dominates both, as we have shown above.

The curves for simple and hybrid hash-join level off at just above 20 megabytes, when a hash table for **R** fits in main memory. It is an easy matter to modify the GRACE hash algorithms so that, if this occurs, then GRACE defaults to the simple algorithm; thus GRACE and simple would be identical above that point.

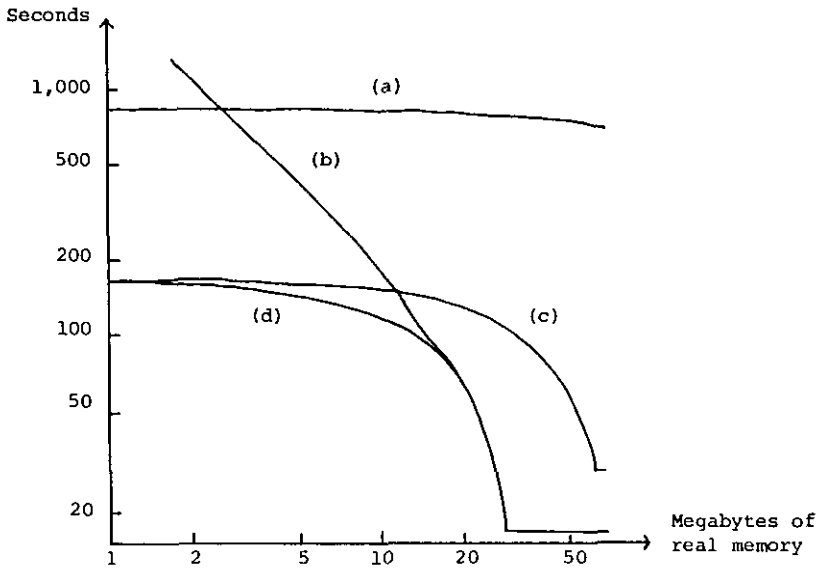


Fig. 7. CPU + I/O times of join algorithms with $|\mathbf{R}| = 20$ megabytes, $|\mathbf{S}| = 40$ megabytes. (a) Sort-merge, (b) simple hash, (c) GRACE hash, (d) hybrid hash. Simple and hybrid hash are identical after 13 megabytes.

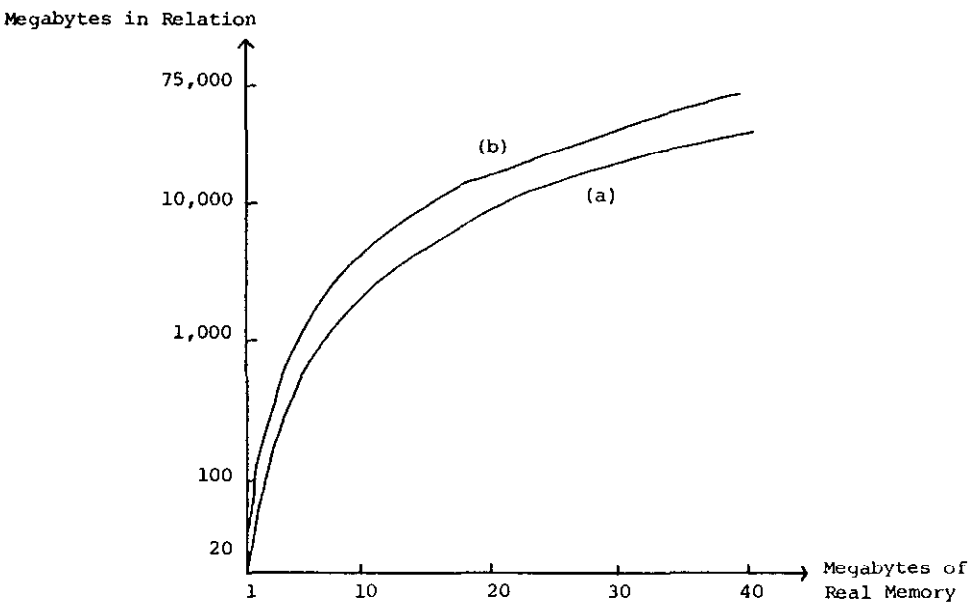


Fig. 8. Maximum relation sizes for varying amounts of main memory. (a) Sort-merge (larger relation), (b) hash-based (smaller relation).

Our algorithms require a minimum number of blocks of real memory, either $\sqrt{|S|}$ (sort-merge) or $\sqrt{F|R|}$ (hash-based algorithms). Therefore, for a given number of blocks of main memory there is a maximum relation size that can be processed by these algorithms. Figure 8 shows these maximum sizes when the

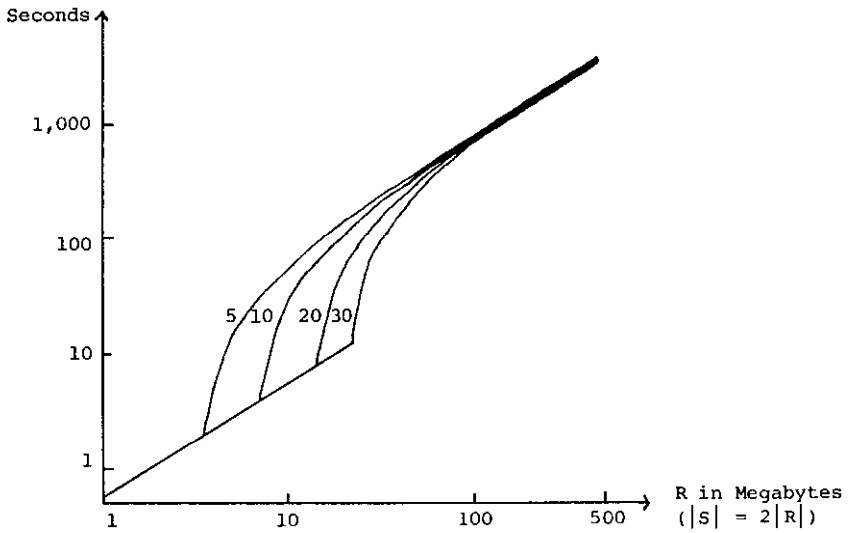


Fig. 9. CPU + I/O time of the hybrid algorithm with 5, 10, 20, and 30 megabytes of real memory.

block size is 25,000 bytes and **F** is 1.4. Note that the sort-merge curve represents the maximum size of the larger relation, while curve (b) shows the maximum size of the smaller relation for the hash-based algorithms.

Figure 9 shows the performance of the hybrid algorithm for a few fixed-memory sizes, as relation sizes vary. Note that when the relation **R** can fit in main memory, the execution time is not very large: less than 12 seconds for relations up to 20 megabytes (excluding, as we always do, the time required to read **R** and **S** and write the result, but assuming no CPU I/O overlap). It is also clear from Figure 9 that when the size of main memory is much less than the size of **R**, performance degrades rapidly.

4. PARTITION OVERFLOW

In all the hashing algorithms that use partitioning, namely simple, GRACE, and hybrid hash, we have made assumptions about the expected size of the subsets of the partitions. For example, in the simple hash-join algorithm, when the relation **R** cannot fit in memory, we assumed that we could choose a hash function h and a partition of its hash values that will partition **R** into two subsets, so that a hash table for the first subset would fit exactly into memory. What happens if we guess incorrectly, and memory fills up with the hash table before we are finished processing **R**?

In [14] this problem is called "bucket overflow." We use the term "partition overflow" because we want to distinguish between the subsets of the partitions of **R** and **S**, produced in the first phase of processing, and the buckets of the hash table of **R** tuples, produced in the second phase, even though both are really hash buckets.

The designers of GRACE deal with overflow by the use of "tuning" (i.e., beginning with very small partitions, and then, when the size of the smaller partitions is known, combining them into larger partitions of the appropriate

size). This approach is also possible in our environment. We present other approaches below.

In all our hash-based algorithms, each tuple of **S** in phase 1 is either used for probing and then discarded, or copied to disk for later use. In phase 2, the remaining tuples on disk are processed sequentially. Since an entire partition of **S** never needs to reside in main memory (as is the case for partitions of **R**), the size of **S**-partitions is of no consequence. Thus we need only find an accurate partitioning of **R**.

To partition **R**, we can begin by choosing a hash function h with the usual randomizing properties. If we know nothing about the distribution of the joining attribute values in **R**, we can assume a uniform distribution and choose a partition of h's hash values accordingly. It is also possible to store statistics about the distribution of h-values. In [16] a similar distribution statistic is studied, where the identity function is used instead of a hash function, and it is shown that using sampling techniques one can collect such distribution statistics on all attributes of a large commercial database in a reasonable time.

Even with the problem reduced to partitioning **R** only, with a good choice of h and with accurate statistics, overflow will occur. In the remainder of this section we show how to handle the three kinds of partition overflow that occur in our algorithms.

4.1 Partition Overflow on Disk

In two algorithms, GRACE and hybrid hash, partitions of **R** are created in disk files, partitions which will later be required to fit in memory. In both algorithms these partitions were denoted $R_1, \ldots, R_n$, where $n = \sqrt{F \mid R \mid}$ for GRACE and $n = \mathbf{B}$ for hybrid. After these partitions are created, it is possible that some of them will be so large that a hash table for them cannot fit in memory.

If one **R** partition on disk overflows, that partition can be reprocessed. It can be scanned and partitioned again, into two pieces, so that a hash table for each will fit in memory. Alternatively, an attempt can be made to partition it into one piece that will just fit, and another that can be added to a partition that turned out to be smaller than expected. Note that a similar adjustment must be made to the corresponding partition of **S**, so that the partitions of **R** and **S** will correspond pairwise to the same hash values.

4.2 Partition Overflow in Memory: Simple Hash

In simple hash, as **R** is processed, a hash table of **R** tuples is built in memory. What if the hash table turns out to be too large to fit in memory? The simplest solution is to reassign some buckets, presently in memory, to the set of "passed-over" tuples on disk, then continue processing. This amounts to modifying the partitioning hash function slightly. Then the modified hash function will be used to process **S**.

4.3 Partition Overflow in Memory: Hybrid Hash

In hybrid hash, as **R** is processed on step 1, a hash table is created from tuples of R_0. What if R_0 turns out to be too large to fit into the memory that remains after some blocks are allocated to output buffers? The solution here is similar to

the simple hash case: reassign some buckets to a new partition on disk. This new partition can be handled just like the others, or it can be spread over others if some partitions are smaller than expected. All this is done before **S** is processed, so the modified partitioning function can be used to process **S**.

5. MEMORY MANAGEMENT STRATEGIES

In this section we consider an alternate memory management strategy for our algorithms. For simplicity we discuss only sort-merge and hybrid hash-join in this section. The behavior of GRACE and simple hash are similar to the behaviors we describe here. We begin in Section 5.1 by describing the weaknesses of the "all real memory" model of the previous sections, where a process was allocated a fixed amount of real memory for its lifetime and the amount of real memory was known to the process. In Section 5.2 we consider virtual memory as an alternative to this simple strategy, with at least a minimum amount of real memory as a "hot set." In Section 5.3 we describe how parts of the data space are assigned to the hot set and to virtual memory, and in Section 5.4 we analyze the impact of this new model on performance. Section 5.5 presents the results of an analytic modeling of performance.

5.1 Problems with an All Real Memory Strategy

Until this section we have assumed a memory management strategy in which each join operation (which we view as a single process) is assigned a certain amount of real memory and that memory is available to it throughout its life. Based on the amount of memory granted by the memory manager (denoted $|\mathbf{M}|$ in Section 2), a strategy will be chosen for processing the join. The key here is knowledge of the amount of memory available. Each of the algorithms we have described above depends significantly on the amount of memory available to the process. There are several problems inherent in designing such a memory manager.

(1) If only one process requests memory space, how much of available memory should be allocated to it? In order to answer this question the memory manager must predict how many and what kind of other processes will require memory before this process completes.

(2) If several processes request memory, how should it be allocated among them? This is probably a simple optimization problem if each process can present to the memory manager an efficiency graph, telling the time the process will take to complete given various possible memory allocations.

(3) If all of memory is taken by active processes, and a new process requests memory, the new process will have to wait until memory is available. This is an intolerable situation in many scenarios. Swapping out a process is not acceptable in general since such large amounts of memory are involved.

As was shown in Figure 8, our algorithms can join huge relations with only a few megabytes of memory. Thus one might argue that the relatively small amounts of real memory needed are affordable in a system with a large main memory. But as one can see from Figures 7 and 9, excellent performance is achieved only when the amount of real memory is close to the size of the smaller

(hybrid) or larger (sort-merge) relation. In general it will not be possible to allocate to each join process an amount of memory near the size of the smaller relation.

5.2 The Hot Set + Virtual Memory Model

One obvious solution to the problems just described is to assign each process all the memory it requests, but in virtual memory, and to let active processes compete for real memory via LRU or some other page-replacement algorithm.

If a process, which is executing a relational operator, is forced to compete for pages with other processes via the usual LRU algorithm, severe thrashing can result. This is pointed out in [17] and in [20], where a variety of relational operators are discussed. Sacco and Scholnick [17] propose to assign each process a certain number of pages (the "hot-set size") which are not subject to demand paging. The hot-set size is estimated by the access planner, and is determined as the point below which a sharp increase in processing time occurs—as the hot-set size varies with each relation and each strategy, it must be estimated by the access planner. Stonebraker [20] proposes allowing the database system to override the usual LRU replacement algorithm when appropriate. We find that a combination of these two approaches best suits our needs.

The algorithms discussed in this paper lend themselves to a hot-set approach since below a certain real memory size ($\sqrt{F \mid R \mid}$ or $\sqrt{\mid S \mid}$) our algorithms behave very poorly.

Therefore, we adopt a similar strategy to that of [17], in that we expect each process to have a certain number of "hot-set" pages guaranteed to it throughout its lifetime. Those hot-set pages will be "wired down" in real memory for the lifetime of the process. A facility for wiring down pages in a buffer is proposed in [9]. The rest of the data space of the process will be assigned to virtual memory. In the next section we describe what is assigned to the hot set and what to virtual memory.

5.3 T and C

Recall that each of the algorithms sort-merge and hybrid hash-join operates in two phases, first processing **R** and **S** and creating either runs or partitions, and secondly reading these runs and partitions and processing them to create the join. Each algorithm's data space also splits into two pieces.

The first piece, which we denote **T** (for Tables), consists of a hash table or a priority queue, plus buffers to input or output the partitions or runs. The second piece of the algorithm's data, which we denote **C** (for Cache), is the partitions or runs generated during phase 1 and read during phase 2.

For sort-merge, as described in Section 2, **T** was fixed in size at $\sqrt{\mid S \mid}$ blocks. If more than $\sqrt{\mid S \mid}$ blocks of real memory were available for sort-merge, the remainder was used to store some or all of **C**, to save I/O costs. The blocks of **C** not assigned to real memory were stored on disk. For hybrid, **T** occupied as much real memory as was available (except that **T** was always between $\sqrt{F \mid R \mid}$ and $\mathbf{F} * \mid \mathbf{R} \mid$ blocks).

Since all of **T** is accessed randomly, and in fact each tuple processed by either algorithm generates a random access to **T**, we assign **T** to the hot set. This means

that the join process needs at least $\sqrt{|S|}$ or $\sqrt{F|R|}$ blocks of real memory to hold **T**. By Figure 8, we can see that $\sqrt{|S|}$ or $\sqrt{F|R|}$ blocks of memory is a reasonable amount. In the case of sort-merge, if additional space is available in the hot set, then some of **C** will be assigned there. For simplicity, henceforth in the case of sort-merge we let **C** refer to the set of runs stored in virtual memory.

For both hybrid and sort-merge join, **C** will be assigned to virtual memory. The hot set + virtual memory model of this section differs from that of the previous sections by substituting virtual memory for disk storage. This allows the algorithms to run with relatively small amounts of wired-down memory, but also to take advantage of other real memory shared with other processes.

To distinguish the algorithms we discuss here from those of Section 2, we append the suffix RM (for Real Memory) to the algorithms of Section 2, which use all real memory, and VM for the variants here, in which **C** is stored in Virtual Memory.

5.4 What Are the Disadvantages of Placing C in Virtual Memory?

Let us suppose that the virtual memory in which **C** resides includes $|\mathbf{C}| * \mathbf{Q}$ blocks of real memory, where $\mathbf{Q} \leq 1$. The quantity **Q** can change during the execution of the algorithm, but for simplicity we assume it to be constant.

What are the potential disadvantages of storing **C** in virtual memory? There are two. The first concerns the blocking factor of one track that we have chosen. In the VM algorithms, where **C** is assigned to virtual memory, during phase 1, in order to take advantage of all $|\mathbf{C}| * \mathbf{Q}$ blocks of real memory, and not knowing what **Q** is, the algorithms should write all $|\mathbf{C}|$ blocks to virtual memory and let the memory manager page out $|\mathbf{C}| * (1 - \mathbf{Q})$ of those blocks. But if the memory manager pages out one page at a time, it may not realize the savings from writing one track at a time. This will result in a higher I/O cost. On the other hand, paging is supported by much more efficient mechanisms than normal I/O. For simplicity, we assume this trade-off results in no net change in I/O costs.

The second possible disadvantage of assigning **C** to virtual memory concerns the usual LRU paging criterion. At the end of phase 1, in the VM algorithms $|\mathbf{C}| * \mathbf{Q}$ blocks of **C** will reside in real memory and $|\mathbf{C}| * (1 - \mathbf{Q})$ blocks on disk. Ideally, all $|\mathbf{C}| * \mathbf{Q}$ blocks will be processed in phase 2 directly from real memory, without having to be written and then read back from disk. As we see below, the usual LRU paging algorithm plays havoc with our plans, paging out many of the $|\mathbf{C}| * \mathbf{Q}$ blocks to disk before they can be processed, and leaving in memory blocks that are no longer of use. This causes a more significant problem. To analyze LRU's behavior more precisely, we must study the access pattern of **C** as it is written to virtual memory and then read back into **T** for processing.

We must estimate how many of the $|\mathbf{C}| * \mathbf{Q}$ blocks in real memory at the end of phase 1 will be paged out before they can be processed. We first consider Hybrid-VM.

In the special case of Hybrid-VM, when $\mathbf{F} * |\mathbf{R}| \leq 2 |\mathbf{M}|$, that is, when only R_0 and perhaps R_1 are constructed, unnecessary paging can be avoided completely. Then $\mathbf{C} = R_1$, or **C** is empty, so **C** consists of one subset and can be read back in phase 2 in any order. In particular, it can be read back in the opposite order from which it was written, thus reading all the in-memory blocks first. Therefore, for

Hybrid-VM, in case $\mathbf{F} \mid \mathbf{R} \mid \leq 2 \mid \mathbf{M} \mid$ and in case the process' resident size does not change, all of the $\mid \mathbf{C} \mid * \mathbf{Q}$ blocks in real memory at the end of phase 1 will be processed before they are paged out.

In the remaining cases of Hybrid-VM, when $\mathbf{F} \mid \mathbf{R} \mid > 2 \mid \mathbf{M} \mid$, there are more than two subsets R_i constructed, that is, $\mathbf{B} > 1$ ($\mathbf{B}$ is defined in Section 2.5). The $\mathbf{B}$ subsets are produced in parallel in phase 1 and read back serially in phase 2. This parallel/serial behavior will cause poor real-memory usage under LRU, as we shall see. To see this, consider the end of phase 1 in Hybrid-VM, which is the time at which all of $\mathbf{C}$ has been created. $\mid \mathbf{C} \mid * \mathbf{Q}$ blocks of $\mathbf{C}$ are in real memory and $\mid \mathbf{C} \mid * (1 - \mathbf{Q})$ blocks are on disk, and the algorithm is about to read $\mathbf{C}$ for processing. After phase 2 has begun and $\mathbf{C}$ has been processed for a while, the part of $\mathbf{C}$ which remains in real memory consists of

C_1: These tuples were on disk at the end of phase 1, and have been read into real memory and processed.
C_2: These tuples were in real memory at the end of phase 1. They have been processed and are no longer needed by the algorithm in phase 2.
C_3: These tuples were in real memory at the end of phase 1. They have not yet been processed in phase 2.

What will happen next, assuming that phase 2 needs to read a block from disk, and therefore to page out a page in memory? If the system uses the usual LRU algorithm, then since the tuples in C_3 were all used less recently than those in C_1 and C_2, the memory manager will choose a page from C_3 to page out, which is exactly the *opposite* of what we would like! This "worst" behavior is pointed out in [20].

Figure 10 gives an intuitive picture of why only $\mid \mathbf{C} \mid * Q^2$ blocks are read directly from real memory in the case we are discussing, namely Hybrid-VM with $\mathbf{B} > 1$. Figure 10 shows the state of the system at the point in phase 2 of Hybrid-VM at which C_3 first becomes empty. After this point all unprocessed tuples of $\mathbf{C}$ are on disk, and so all requests for tuples from $\mathbf{C}$ will cause a page fault. Before this point requests for tuples from $\mathbf{C}$ might not cause a page fault, if those tuples were in real memory at the beginning of phase 2. The set D_1 denotes the location of the tuples of C_1 before they were read onto disk. Since the number of bytes in C_1 and D_1 are equal, a little algebra shows that x must be $\mathbf{Q} * \mathbf{B}$, and then that there are $\mid \mathbf{C} \mid * Q^2$ blocks in C_2. Thus only $\mid \mathbf{C} \mid * Q^2$ blocks have been read directly from real memory.

This argument is based on the ideal picture of Figure 10. In practice, the sets C_1, D_1, and C_2 in Figure 10 have jagged edges, and the argument we have given is not precise. However, it can be shown that this argument is valid if $\mathbf{B}$ is large and if the subsets are created at uniform speed, by the partitioning process in phase 1. This analysis indicates that in Hybrid-VM, at least when $\mathbf{B}$ is large, only $\mid \mathbf{C} \mid * Q^2$ blocks are read directly from real memory.

A similar analysis is valid for sort-merge, based on the fact that runs in sort-merge are produced serially and read back in parallel, with a similar $\mid \mathbf{C} \mid * Q^2$ conclusion.

Is there a way to avoid this poor paging behavior? One relatively simple technique, called "throw immediately" in [20], is to mark a page of $\mathbf{C}$ as "aged'

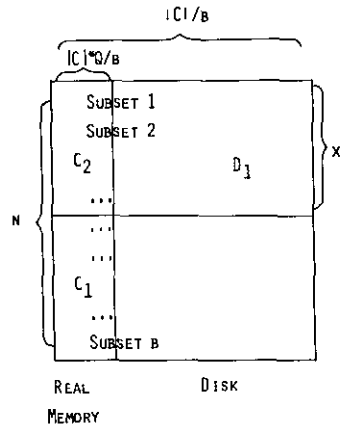


Fig. 10. Hybrid hash-join in virtual memory with LRU.

after it has been read into **T**, so that when part of **C** needs to be paged out the system will take the artificially aged page instead of a yet unprocessed page. With this page-aging facility, a full $|\mathbf{C}| * \mathbf{Q}$ blocks of **C** will be read directly from real memory and will generate I/O savings.

5.5 Performance in the Hot Set + Virtual Memory Model

Figure 11 presents the results of an analytic modeling of Hybrid-VM, assuming that 5 megabytes of real memory are allocated to the hot set where **T** resides, and other real memory is used to support the virtual memory in which **C** resides. In graph (a), we have assumed that only $|\mathbf{C}|\ Q^2$ blocks of $|\mathbf{R}|$ are read directly from real memory, as would be the case if LRU were used. In (b) we assume $|\mathbf{C}| * Q$ blocks of **C** are read from real memory, as would be the case if page-aging were used.

According to Figure 11, the most efficient processing, with all real memory, requires 28 megabytes and takes 16 seconds, compared to 54 megabytes and 25 seconds when virtual memory is used, with the hot-set size of 5 megabytes. More memory is needed when virtual memory is used because in that case subsets from both **R** and **S** are stored.

One way to explain the poorer performance of graph 11(b) compared to graph 11(c) is to view 11(b) as the result of reducing the size of the hot set, and therefore **T**. When the hot set is large (e.g., when it can hold a hash table for **R**) no virtual memory is needed, and performance is given by (c) at its minimum CPU + I/O time. As the hot-set size decreases, performance degrades. At the minimum, when the hot-set size is $\sqrt{F\,|R|}$, hybrid's performance in the hot set + virtual memory model is identical to that of GRACE, since the algorithms for GRACE and hybrid in Section 2 are identical when $|\mathbf{M}| = \sqrt{F\,|R|}$.

The performance of sort-merge in the hot set + virtual memory model with LRU plus page-aging is identical to the all real memory case, because sort-merge uses real memory, beyond the $\sqrt{|S|}$ blocks needed for **T** to store **C** and save I/O. Therefore, only $\sqrt{|S|}$ blocks of real memory should be assigned to the hot set for sort-merge.

We conclude that if page-aging is possible, then the performance of sort-merge is unaffected in the hot set + virtual memory model, but the performance of the

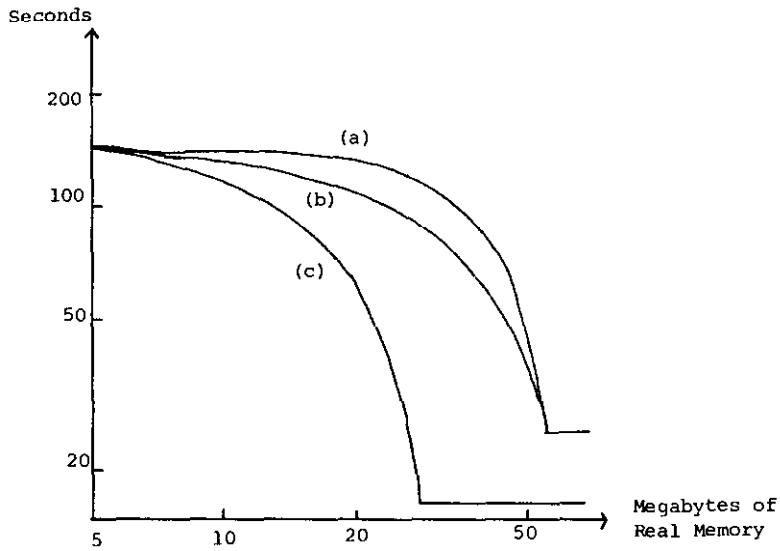


Fig. 11. CPU + I/O time of Hybrid algorithm for varying amounts of memory. $|\mathbf{R}| = 20$ megabytes, $|\mathbf{S}| = 40$ megabytes. For (a) and (b), hot-set size is 5 megabytes. (a) Hybrid-VM with LRU, (b) Hybrid-VM with page-aging, (c) Hybrid-RM: all real memory.

hybrid join degrades, as the hot-set size decreases, to the performance of GRACE. Since we have shown, in Section 3, that GRACE typically dominates sort-merge, we conclude that hybrid typically dominates sort-merge, even in the hot set + virtual memory model.

6. OTHER TOOLS

In this section we discuss three tools that have been proposed to increase the efficiency of join processing, namely database filters, Babb arrays, and semijoins. Our objective is to show that all of them can be used equally effectively with any of our algorithms.

Database filters [19] are an important tool to make database managers more efficient. Filters are a mechanism to process records as they come off the disk, and send to the database only those which qualify. Filters can be used easily with our algorithms, since we have made no assumption about how the selections and projections of the relations **R** and **S** are made before the join.

Another popular tool is the Babb array [1]. This idea is closely related to the concept of partitioning which we have described in Section 2. As **R** is processed, a boolean array is built. Each bit in the array corresponds to a hash bucket, and the bit is turned on when an **R** tuple hashes into that bucket. Then, as each tuple s from **S** is processed, the boolean array is checked, and if s falls in a bucket for which there are no **R** tuples, the s tuple can be discarded without checking **R** itself. This is a very powerful tool when relatively few tuples qualify for the join.

The Babb array can easily be added to any of our algorithms. The first time **R** is scanned, the array is built, and when **S** is scanned, some tuples can be discarded. Its greatest cost is for space to store the array. Given a limited space in which to

store the array, another problem is to find a hash function to use in constructing the array so that the array will carry maximum information. It is possible to use several hash functions and an array for each, to increase the information, but with limited space this alternative allows each hash function a smaller array and therefore less information. Babb arrays are most useful when the join has a high selectivity (i.e., when there are few matching tuples).

Finally, we discuss the semijoin [2]. This is often regarded as an alternative way to do joins, but as we shall see it is a special case of a more general tool. The semijoin is constructed as follows.

(1) Construct the projection of **R** on its joining attributes. We denote this projection by $\pi(R)$.
(2) Join $\pi(R)$ to **S**. The result is called the semijoin of S with R and is denoted $S \ltimes R$. The semijoin of S with R is the set of **S** tuples that participate in the join of **R** and **S**.
(3) Join $S \ltimes R$ to **R**. The result is equal to the join of **R** and **S**.

These steps can be integrated into any of our algorithms. When first scanning **R**, one constructs $\pi(R)$ and, when first scanning **S**, one discards tuples whose joining attribute values do not appear in $\pi(R)$. If the join has a low selectivity, then this will reduce significantly the number of **S** tuples to be processed, and will be a useful tool to add to any of the above algorithms.

The most significant expense of the semijoin tool is space to store $\pi(R)$. For example, in some cases $\pi(R)$ might be almost as large as **R**. Can we minimize the space needed to store $\pi(R)$? One obvious candidate is a Babb array. In fact, the Babb array and semijoins are just two specific examples of a more general tool, which can be described as follows.

(1′) Construct a structure $\sigma(R)$ which contains some information about the relation $\pi(R)$, where $\pi(R)$ is defined in (1) above. In particular, $\sigma(R)$ must contain enough information to tell when a given tuple is *not* in $\pi(R)$.
(2′) Scan **S** and discard those tuples which, given the information in $\sigma(R)$, cannot participate in the join. Denote the set of undiscarded **S** tuples by $\mathbf{R} \Sigma \mathbf{S}$.
(3′) Join $\mathbf{R} \Sigma \mathbf{S}$ to **R**. The result is equal to the join of **R** with **S**.

The semijoin tool takes $\sigma(R)$ equal to $\pi(R)$, while the Babb array is another representation of $\sigma(R)$, which may be much more compact than $\pi(R)$. This more general tool is a special case of the Tuneable Dynamic Filter described in [13].

7. CONCLUSIONS

We have defined and analyzed three hash-based equijoin algorithms, plus a version of sort-merge that takes advantage of significant amounts of main memory. These algorithms can also operate efficiently with relatively little main memory. If the relations are sufficiently large, then one hash-based algorithm, a hybrid of the other two, is proved to be the most efficient of all the algorithms we study.

The hash-based join algorithms all partition the relations into subsets which can be processed in main memory. Simple mechanisms exist to minimize overflow of these partitions and to correct it when it occurs, but the quantitative effect of these mechanisms remains to be investigated.

The algorithms we describe can operate in virtual memory with a relatively small "hot set" of nonpageable real memory. If it is possible to age pages, marking them for paging out as soon as possible, then sort-merge has the same performance in the hot set plus virtual memory model as in the all real memory model, while the performance of the hybrid algorithm degrades. If aging is not possible, then the performance of both hybrid and sort-merge degrades. In fact, if a fraction $\mathbf{Q}$ of the required virtual memory space is supported by real memory, then the absence of an aging facility can result in performance equal to that with only a fraction Q^2 of real pages. In the hot set plus virtual memory model, the hybrid hash-based algorithm still has better performance than sort-merge for sufficiently large relations.

Database filters, Babb arrays, and semijoin strategies can be incorporated into any of our algorithms if they prove to be useful.

We conclude that, with decreasing main memory costs, hash-based algorithms will become the preferred strategy for joining large relations.

REFERENCES

1. BABB, E. Implementing a relational database by means of specialized hardware. *ACM Trans. Database Syst. 4*, 1 (Mar. 1979).
2. BERNSTEIN, P. A. Query processing in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 6*, 4 (Dec. 1981), 602–625.
3. BITTON, D., BORAL, H., DEWITT, D., AND WILKINSON, W. Parallel algorithms for the execution of relational database operations. *ACM Trans. Database Syst. 8*, 3 (Sept. 1983), 324–353.
4. BLASGEN, M. W., AND ESWARAN, K. P. Storage and access in relational databases. *IBM Syst. J. 16*, 4 (1977).
5. BRATBERGSENGEN, K. Hashing methods and relational algebra operations. In *Proceedings of the Conference on Very Large Data Bases* (Singapore, 1984).
6. DEWITT, D., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBRAKER, M., AND WOOD, D. Implementation techniques for main memory database systems. In *Proceedings of SIGMOD* (Boston, 1984), ACM, New York.
7. DEWITT, D., AND GERBER, R. Multiprocessor hash-based join algorithms. In *Proceedings of the Conference on Very Large Data Bases* (Stockholm, 1985).
8. DIGITAL EQUIPMENT CORP. Product announcement, 1984.
9. EFFELSBERG, W., AND HARDER, T. Principles of database buffer management. *ACM Trans. Databse Syst. 9*, 4 (Dec. 1984), 560–595.
10. GARCIA-MOLINA, H., LIPTON, R., AND VALDES, J. A massive memory machine. *IEEE Trans. Comput. C-33*, 5 (1984), 391–399.
11. GOODMAN, J. R. An investigation of multiprocessor structures and algorithms for data base management. Electronics Research Lab. Memo. ECB/ERL M81/33, Univ. of California, Berkeley, 1981.
12. KERSCHBERG, L., TING, P., AND YAO, S. Query optimization in Star computer networks. *ACM Trans. Database Syst. 7*, 4 (Dec 1982), 678–711.
13. KIESSLING, W. Tuneable dynamic filter algorithms for high performance database systems. In *Proceedings of the International Workshop on High Level Computer Architecture* (May 1984), 6.10-6.20.
14. KITSUREGAWA, M., ET AL. Application of hash to data base machine and its architecture. *New Generation Comput. 1* (1983), 62–74.

15. KNUTH, D. *The Art of Computer Programming: Sorting and Searching, Vol. 3.* Addison-Wesley, Reading, Mass., 1973.
16. PIATETSKY-SHAPIRO, G., AND CONNELL, C. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of SIGMOD Annual Meeting* (Boston, 1984), ACM, New York.
17. SACCO, G. M., AND SCHOLNICK, M. A mechanism for managing the buffer pool in a relational database system using the hot-set model. Computer Science Res. Rep. RJ-3354, IBM Research Lab., San Jose, Calif., Jan. 1982.
18. SEVERANCE, D., AND DUEHNE, R. A practitioners guide to addressing algorithms. *Commun. ACM 19*, 6 (June 1976), 314–326.
19. SLOTNICK, D. Logic per track devices. In *Advances in Computers*, Vol. 10, J. Tou, Ed., Academic Press, New York, 1970, 291–296.
20. STONEBRAKER, M. Operating system support for database management. *Commun. ACM 24*, 7 (July 1981), 412–418.
21. VALDURIEZ, P., AND GARDARIN, G. Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. Database Syst. 9*, 1 (Mar. 1984), 133–161.
22. YAMANE, Y. A hash join technique for relational database systems. In *Proceedings of the International Conference on Foundations of Data Organization* (Kyoto, May 1985).

Received August 1984; revised December 1985; accepted December 1985.