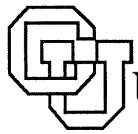
Five Performance Enhancements for Hybrid Hash Join

Goetz Graefe

CU-CS-606-92 July 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Five Performance Enhancements for Hybrid Hash Join

Goetz Graefe
University of Colorado at Boulder

Abstract

In this paper, we focus on set matching algorithms such as intersection, difference, union, and relational join, using join as a representative for all these matching problems. We discuss five performance enhancements for hybrid hash join algorithms, namely data compression, large cluster sizes and multi-level recursion, role reversal of build and probe inputs, histogram methods to exploit non-uniform data and hash value distributions (skew), and join algorithms for multiple inputs. While each of the enhancements is fairly simple, the most surprising result is that hash value skew can be exploited and improve performance rather than being a danger to hybrid hash join performance as conventionally thought. Our design for hash-based N-way matching algorithms is a dual to pipelining data without intermediate sorting between multiple merge-joins on the same attribute (interesting orderings), and exceeds its performance advantages.

Each of the performance enhancements can be used by itself or they can be combined with each other as well as with parallel query execution techniques. The cumulative effect of the optimizations is that hybrid hash join will almost always be the set matching algorithm of choice, even in situations for which earlier research had recommended sorting and merge-join.

Index Terms

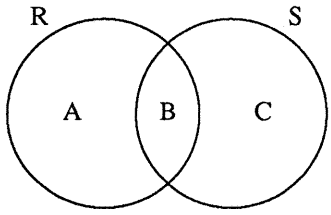
Database Query Processing, Set Matching, Hybrid Hash Join, Performance, Tuning, Data Compression, I/O Speed, Fan-Out, Recursion Depth, Role Reversal, Non-Uniformity, Histograms, Interesting Orderings, N-Way Partitioning.

1. Introduction

Database management systems will continue to manage large data volumes. Thus, efficient algorithms for accessing and manipulating large sets and sequences will be required to provide competitive performance. The advent of object-oriented and extensible database systems will not solve this problem; on the contrary, modern data models exacerbate it. In order to manage complex objects as efficiently as today's database systems manage simple records, query processing algorithms and their performance must be explored and improved.

In this paper, we focus on the performance of hybrid hash join and explore a number of performance enhancements that we have found to be very effective. While each of the enhancements is quite simple to understand and to implement, the cumulative effect of the optimizations is that hybrid hash join will almost always be the set matching algorithm of choice. This will be true even in the situations for which earlier research such as our comparison of sorting and hashing [18] had recommended sorting and merge-join, namely (i) the presence or danger of hash value skew (including skew created by duplicate data values), (ii) the query optimizers' inability to determine the inputs' relative sizes a priori in complex queries, and (iii) complex query predicates using the same join attribute, i.e., queries that permit exploiting "interesting orderings" [33] by pipelining intermediate results from one merge-join to the next without sorting the intermediate result. Moreover, all the performance enhancements presented here can be freely combined with parallel query execution techniques on shared-, distributed-, and hierarchical-memory architectures.

Both in the paper's title and in the discussion, we use relational join as a representative for a number of important database operations, although all issues and effects are equally applicable to other operations frequently used in database query processing. These operations are called the binary matching operations here. The most prominent among these operations is the relational join; the other operations are left and right semi-join, left, right, and symmetric outer-join, anti-join, intersection, union, left and right difference, and anti-difference. Figure 1 shows the basic principle underlying all these operations, namely separation of the matching and non-matching components of two sets, called R and S in the figure, and production of appropriate subsets, possibly after some transformation and combination of items as in the case of a join. All these operations require basically the same steps and can be implemented with the same algorithms. In particular, there is a hybrid hash join variant for each of these operations. For simplicity, however, only join algorithms are discussed here. Furthermore, we only discuss algorithms for one join attribute since the algorithms and their performance for multi-attribute joins are not different.



Output	Match on all Attributes	Match on some Attributes
A	Difference	Anti-semi-join
B	Intersection	Join, semi-join
C	Difference	Anti-semi-join
A, B		Left outer join
A, C	Symmetric difference	Anti-join
B, C		Right outer join
A, B, C	Union	Symmetric outer join

Figure 1. Binary One-to-One Matching.

Since any data model supporting sets and lists requires at least intersection, union, and difference operations, we believe that this discussion is relevant to relational, extensible, and object-oriented database systems alike. Moreover, binary matching problems occur in some surprising places. Consider an object-oriented database system that uses a table to map logical object identifiers (OID's) to physical locations (record identifiers or RID's). Resolving a set of OID's to RID's can be regarded (as well as optimized and executed) as a semi-join of the mapping table and the set of OID's, and all conventional join strategies can be employed. Another example that can occur in a database management system for any data model is the use of multiple indices in a query: the pointer (OID or RID) lists obtained from the indices must be intersected (for a conjunction) or unioned (for a disjunction) to obtain the list of pointers to items that satisfy the whole query. Furthermore, the actual lookup of the items using the pointer list can be regarded as a semi-join of the underlying data collection (such as the disk) and the list, as in Kooi's thesis and the Ingres product [25, 26] and in a recent study by Shekita and Carey [36]. Thus, even if relational systems were completely replaced by object-oriented database systems, set matching and join techniques developed in the relational context would continue to be important for the performance of database systems.

In the next section, we discuss hash-based query processing algorithms including hybrid hash join and recursion. The following five sections explore five performance enhancement, namely data compression, large cluster sizes and multi-level recursion, role reversal of build and probe inputs, histogram methods to exploit non-uniform data and hash value distributions (skew), and join algorithms for multiple inputs. In the last section, we summarize our ideas and results, offer our conclusions from this research, and point to interesting and relevant future research issues.

2. Hash-Based Database Query Processing

In this section[1], we review hash-based query processing algorithms, including hybrid hashing, partitioning with multiple recursion levels, special characteristics of binary operations, and cost functions. The cost functions are actually is not necessary for execution, because the hybrid hash join algorithm adapts to its input size, but they are useful for cost estimation during query optimization and for our comparisons here.

For the I/O cost formulas given in this paper, we assume that the left and right inputs have R and S pages, respectively, and that the memory size is M pages. We omit the cost of reading stored inputs and writing the final outputs from the cost formulas; in other words, we only consider I/O to temporary files. Furthermore, we

[1] Much of this section has been derived from [16].

frequently calculate the data amount that must be read or written rather than the number of I/O operations or the I/O time.

Most of today's database management systems use only nested loops and merge-join, because an analysis performed in connection with the System R project determined that of all the join methods considered, one of these two always provided either the best or very close to the best performance [2, 3]. However, the System R study did not consider hash join algorithms, which are now regarded as more efficient in many cases.

Hashing is an alternative to sorting for many matching tasks, not only for the binary matching problems shown in Figure 1. Other database problems for which hash algorithms have been devised include aggregation, duplicate removal, and relational division [16]. In general, when equality matching is required, hashing should be considered, because the complexity of set algorithms based on hashing is $O(N)$ rather than $O(N \log N)$ as for sorting.

Principles of Hash-Partitioning Algorithms

Hash-based query processing algorithms use an in-memory hash table of database objects to perform their matching task. If the entire hash table (including all records or items) fits into memory, hash-based query processing algorithms are very easy to design, understand, and implement, and outperform sort-based alternatives. Note that for binary matching operations (such as join or intersection) only one of the two inputs must fit into memory. However, if the required hash table is larger than memory, *hash table overflow* occurs and must be dealt with.

In order to manage hash table overflow, the input is divided into multiple partition files such that partitions can be processed independently from one another and the concatenation of the results of all partitions is the result of the entire operation. Partitioning should ensure that the partitioning files are of roughly even size, and can be done using either hash-partitioning or range-partitioning, i.e., based on keys estimated to be quantiles. Usually, partition files can be processed using the original hash-based algorithm. Some output buffer space is required for each partition being written, namely the size of one unit of I/O, called *cluster* with size C in this paper. The maximal partitioning *fan-out* F, i.e., number of partition files created, is determined by the memory size divided by the cluster size, i.e., $F = \lfloor M / C \rfloor$ just like the fan-in for sorting.

There are basically two control strategies for managing hash table overflow, namely *avoidance* and *resolution*. In hash table overflow avoidance, the input set is partitioned into F partition files before any in-memory hash table is built. If it turns out that fewer partitions than have been created would have been sufficient to obtain partition files that will fit into memory, bucket tuning (collapsing multiple small buckets into larger ones) and dynamic destaging (determining which buckets should stay in memory) can improve the performance of hash-based operations [23, 28].

Hash table overflow resolution starts with the assumption that overflow will not occur, but resorts to

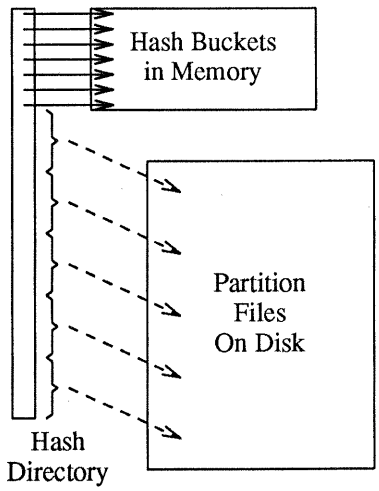


Figure 2. Hybrid Hashing.

basically the same set of mechanisms as hash table overflow avoidance once it does occur. Hybrid hashing methods combine the two ideas [8, 35]. They start out with the premise that no overflow will occur; if it does, however, they partition the input into multiple partitions of which only one is written immediately to temporary files on disk. The other $F - 1$ partitions remain in memory. If another overflow occurs, another partition is written to disk. If necessary, all F partitions are written to disk. Thus, hybrid hash algorithms use all available memory for in-memory processing, but at the same time are able to process large inputs by overflow resolution. Figure 2 shows the idea of hybrid hash algorithms. As many hash buckets as possible are kept in memory, e.g., as linked lists as indicated by solid arrows. The other hash buckets are spooled to temporary disk files, called the overflow or partition files, and are processed in later stages of the algorithm. Hybrid hashing is applicable if the input size I is larger than memory but smaller than the memory size multiplied by the fan-out, i.e., $M < I \leq F \times M$.

In order to predict the number of I/O operations, the number of required partition files on disk must be determined. Call this number K, which must satisfy $0 \leq K \leq F$. Presuming that the assignment of buckets to partitions is optimal and each partition file will be of size M, the amount of data that can be written to all K partition files is equal to $K \times M$. Writing K partition files requires $K \times C$ output buffer space, leaving $M - K \times C$ memory for the hash table. The optimal K for a given input size I is the minimal K for which $K \times M + M - K \times C \geq I$. Solving this inequality and taking the smallest such K results in $K = \lceil (I - M) / (M - C) \rceil$. The minimal possible I/O cost, including a factor of 2 for writing and reading the partition files and measured in the amount of data that must be written or read, is $2 \times (I - M + K \times C)$. To determine the I/O time, this amount must be divided by the cluster size and multiplied with the I/O time for one cluster.

For example, consider an input of $I = 240$ pages, a memory of $M = 80$ pages, and a cluster size of $C = 8$ pages. The fan-out is $F = \lfloor 80 / 8 \rfloor = 10$. The number of partition files that need to created on disk is $K = \lceil (240 - 80) / (80 - 8) \rceil = 3$. In other words, in the best case, $K \times C = 3 \times 8 = 24$ pages will be used as output buffers to write $K = 3$ partition files of no more than $M = 80$ pages, and $M - K \times C = 80 - 3 \times 8 = 56$ pages of memory will be used as hash table. The total amount of data written to and read from disk is $2 \times (240 - 80 + 3 \times 8) = 368$ pages. If writing or reading a cluster of $C = 8$ pages takes 30 msec, the total I/O time is $368 / 8 \times 30 = 1{,}380$ msec.

In the calculation of K, we assumed an optimal assignment of hash buckets to partition files. If buckets were assigned in the most straightforward way, e.g., by dividing the hash directory into F equal-size regions and assigning the buckets of one region to a partition as indicated in Figure 2, all partitions will be of nearly the same size and either all or none of them will fit into memory. In other words, once hash table overflow occurs, all input will be written to partition files. Thus, we presumed in the earlier calculations that hash buckets were assigned optimally to partitions. There are two good ways to assign hash buckets to partitions. First, in bucket tuning, a large number of small partition files is created and then collapsed into fewer partition files no larger than memory. In the example, three partitions of 24 pages would be read back into memory after the remaining seven partitions had been collapsed into three partitions of no more than $M = 80$ pages. Bucket tuning is not effective in unary operations such as aggregation and duplicate removal; however, in binary operations such as intersection and relational join, it avoids writing parts of the second (typically larger) input to disk because the partitions in memory can be matched immediately using a hash table in the memory not required as output buffer because a number of small partitions have been collapsed into fewer, larger partitions. Second, statistics gathered before hybrid hashing commences can be used to assign hash buckets to partitions, as will be discussed later in this paper.

Recursive Hash-Partitioning

Unfortunately, it is possible that one or several partition files are larger than memory. In that case, partitioning is used recursively until the file sizes have shrunk to memory size or at least until hybrid hashing applies. Figure 3 shows how a hash-based algorithm for a unary operation such as aggregation or duplicate removal partitions its input over multiple recursion levels. The recursion terminates when the partition files fit into memory. In the deepest recursion level, hybrid hashing may be employed.

If the partitioning (hash) function is good and creates a uniform hash value distribution, the file size in each recursion level shrinks by a factor equal to the fan-out, and therefore the number of recursion levels L is logarithmic with the size of the input being partitioned. After L partitioning levels, each partition file is of size $II = I / F^L$. In order to obtain partition files suitable for hybrid hashing (with $M < II \leq F \times M$), the number of full recursion levels L, i.e., levels at which hybrid hashing is not applied, is $L = \left\lfloor \log_F (I / M) \right\rfloor$. The I/O cost of the remaining step using hybrid hashing can be estimated by using the hybrid hash cost formula above with I

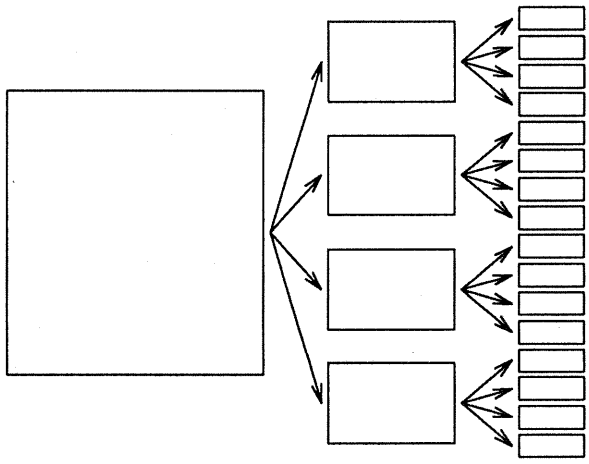


Figure 3. Recursive Partitioning.

replaced by II and multiplying the cost with F^L because hybrid hashing is used for many partition files. Thus, the total I/O cost for partitioning an input and using hybrid hashing in the deepest recursion level is

$$2 \times I \times L + 2 \times F^L \times \left(II - M + K \times C \right)$$

$$= 2 \times \left(I \times (L+1) - F^L \times (M - K \times C) \right)$$

$$= 2 \times \left(I \times (L+1) - F^L \times \left(M - \lceil (II - M) / (M - C) \rceil \times C \right) \right).$$

Since data items are assigned to partitions based on their hash values, using the same hash function in the next recursion level would not be effective. Thus, the hash function at each recursion level is different from the hash functions used in earlier recursion levels. Universal hash functions provide sets of hash functions that are in some sense "orthogonal" to each other and therefore are excellent choices for successive recursion levels [6].

Join and Other Binary Matching Operations

For binary matching operations such as join and union, hash-based algorithms are based on the idea of building an in-memory hash table on one input (the smaller one, usually called the *build input*) and then probing this hash table using items from the other input (usually called the *probe input*). These algorithms have only recently found greater interest [4, 8-10, 14, 22, 23, 28, 29, 31, 35, 38]. One reason is that they work very fast, i.e., without any temporary files, if the build input does indeed fit into memory, independently of the size of the probe input. However, they require overflow avoidance or resolution methods for larger build inputs, and suitable methods were developed and experimentally verified only in the mid-1980s, most notably in connection with the Grace and Gamma database machine projects [10, 11, 14, 22]

In hash-based join methods, build and probe inputs are partitioned using the same partitioning function, e.g., the join key value modulo the number of partitions. The final join result can be formed by concatenating the join results of pairs of partitioning files. Figure 4, adapted from a similar diagram in [22], shows the effect of partitioning the two inputs of a binary operation such as join into hash buckets and partitions. Without partitioning, each item in the first input must be compared with each item in the second input; this would be represented by complete shading of the entire diagram. With partitioning, items are grouped into partition files, and only pairs in the series of small rectangles (representing the partitions) must be compared.

If a build partition file is still larger than memory, recursive partitioning is required. Recursive partitioning is used for both build and probe partitioning inputs using the same hash and partitioning functions. Figure 5 shows how both inputs are partitioned together. The partial results obtained from pairs of partition files are concatenated to form the result of the entire match operation. Recursive partitioning stops when the build partition fits into

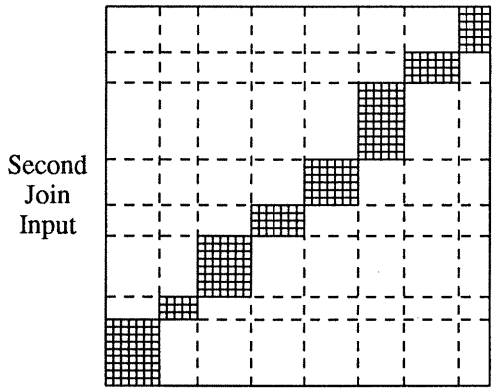


Figure 4. Effect of Partitioning for Join Operations.

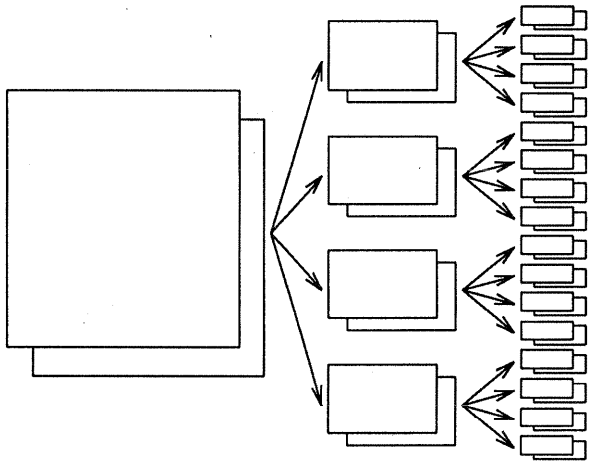


Figure 5. Recursive Partitioning in Binary Operations.

memory. Thus, the recursion depth of partitioning for binary match operators depends only on the size of the build input (which therefore should be chosen to be the smaller input) and is independent of the size of the probe input. Compared to sort-based binary matching operators, i.e., variants of merge-join in which the number of merge levels is determined for each input individually, hash-based binary matching operators are particularly effective when the input sizes are very different [4, 18].

The I/O cost for binary hybrid hash operations can be determined by the number of complete levels (i.e., levels without hash table) and the fraction of the input remaining in memory in the deepest recursion level. For memory size M, cluster size C, partitioning fan-out $F = \lfloor M / C \rfloor$, build input size R, and probe input size S, the number of complete levels is $L = \left\lfloor \log_F (R / M) \right\rfloor$, after which the build input partitions should be of size $R' = R / F^L$. The I/O cost for the binary operation is the cost of partitioning the build input divided by the size of the build input and multiplied by the sum of the input sizes. Using the cost formula for unary hashing discussed earlier, the total amount of I/O for a recursive binary hash operations is

$$2 \times \left(R \times (L+1) - F^L \times \left(M - \lceil (R' - M) / (M - C) \rceil \times C \right) \right) / R \times (R + S),$$

which can be approximated with

$$2 \times \log_F (R / M) \times (R + S).$$

In other words, the cost of binary hash operations on large inputs is logarithmic; the main difference to the cost of sorting and merge-join is that the recursion depth (the logarithm) depends only on one input, the build input, and is not taken for each input individually.

3. Data Compression

Having discussed the use of binary matching in any data model, the similarity of a number of binary matching problems, and hybrid hash join as a representative of hash-based binary matching algorithms, we now explore a series of five performance enhancements to hybrid hash join. In this section, we outline how data compression can be exploited in database systems beyond its obvious advantages, how the amount of data that needs to be decompressed can be minimized, and how standard query processing algorithms can be adapted to work on compressed data.

The compression rates that can be achieved for any dataset depend, of course, on the attribute types and value distributions. For example, it is difficult to compress binary floating point numbers, but relatively easy to compress text by a factor of 2 to 3 [1, 27]. In the following, we do not require that all data is text; we only require that some compression can be achieved. Since text attributes tend to be the largest fields in database files, we suspect that expecting an overall compression factor of 2 is realistic for many or even most database files. Optimal performance can only be obtained by judicious decisions which attributes to compress and which compression method to employ.

Query Processing with Compressed Values

For database query processing, we suggest that a fixed compression scheme be used for each attribute. The scheme should be fixed for each attribute to permit comparisons of database values with (compressed) predicate constants. Actually, it should be fixed for each domain, not each attribute, because this will allow comparing compressed values from different sources, e.g., department numbers in the employee and the department relations. This is an important new idea for employing data compression in database management, because compressing all values of a domain with the same compression encoding permits performing a number of frequently used operations without decompression, namely all operations based on equality comparisons. In other words, instead of decompressing stored data as soon as they are loaded into memory, we recommend keeping data compressed throughout query processing (or at least as long as possible) and decompressing data only when absolutely required. Decompression is not required for equality comparisons, only for ordered (e.g., "≤") comparisons, arithmetic, and presentation to a user or an application program. Thus, the effort required for decompression is typically much less than the effort required in schemes which use compression only for storage and disk bandwidth, but not during processing. We believe that decompressing basically only query results is an acceptable tradeoff against all the benefits of compression.

Using only one compression scheme per domain is facilitated by the move in modern database management systems towards the encapsulation of data types (and classes) in abstract data types (ADT's). A fixed compression scheme for each ADT does not rule out dynamic compression schemes, even if it seems to on first sight. Instead of adjusting the encoding all the time, e.g., after each character as can be done in dynamic compression and decompression of transmission streams, the compression encoding for efficient query processing can only be adjusted during database reorganization. Suitable statistics can be gathered while unloading the database, and a new encoding can be used starting when the database is reloaded. In fact, separating statistics gathering for dynamic compression schemes during unloading and compression of data during reloading eliminates the start-up and adjustment period during which dynamic compression schemes do not work very effectively [1]. The parameters of the compression scheme used during reloading are made part of the meta-data or catalogs similar to the size and distribution data stored in today's relational catalogs.

Compression can be exploited far beyond improved I/O performance in database query processing. First, exact-match comparisons can be performed on compressed data, both in scans comparing a compressed constant with compressed database values and in exact-match index lookup. Second, projection and duplicate removal can be done without decompressing data, since equal uncompressed records will have equal compressed images. Although the algorithms used for aggregation and duplicate removal are principally the same, aggregation requires that attributes on which arithmetic (minimum, sum, average) is performed typically must be decompressed. Third, attributes can remain compressed for binary matching operations. Since we require that compression schemes be fixed for each domain, a join on compressed key values will give the same results as a join on normal,

decompressed key values. It might seem unusual to perform a merge-join in the order of compressed values, but it nonetheless is possible and produces correct results. The same arguments as for join hold for semi-join, outer-join, union, intersection, and difference.

Performance Observations

In order to see the performance effects of compression on database query processing and in particular on hybrid hash join, consider the I/O costs for hybrid hash join using $M = 400$ pages of memory of two inputs with the uncompressed sizes $R = 1{,}000$ and $S = 5{,}000$ pages, or half as much compressed. For simplicity, we ignore fragmentation and assume uniform hash value distributions. Because the time for reading the original inputs depends on the compression effectiveness, we have included this I/O in the cost calculations in this section.

First, consider the cost of hybrid hash join using uncompressed data. Since recursion will not be required (recursion depth $L = 0$), the amount of I/O is

$$2 \times \left\lfloor 1000 - 400 + \lceil (1000 - 400) / (400 - 1) \rceil \right\rfloor / 1000 \times (1000 + 5000) = 7224$$

pages I/O's for temporary files plus 6,000 I/O's on permanent files, for a total of 13,224 I/O's. Now consider joining compressed inputs. The I/O for partition files is

$$2 \times \left\lfloor 500 - 400 + \lceil (500 - 400) / (400 - 1) \rceil \right\rfloor / 500 \times (500 + 2500) = 1212$$

pages for temporary files and 3,000 pages for the input files, for a total of 4,212 I/O's.

The total I/O costs differ by a factor of more than three, 13,224 vs. 4,212 I/O's. While the I/O costs for the permanent files differ by a factor of two, as expected for a volume reduction to 50%, the I/O costs for temporary files differ by a factor of almost six. A factor of two could easily be expected; however, the improved utilization of memory (more records remain in the hash table during the build phase) significantly reduces the number of records that must be written to overflow files. Thus, compression reduces both the number and the size of records written to temporary files, resulting in a reduction of I/O costs on temporary files by a factor of six.

If the compression scheme had been a little more effective, i.e., a factor of 2½ instead of 2 or a reduction to 40% instead of 50%, overflow files would have been avoided entirely for compressed data, leaving only the I/O on permanent data. The total I/O costs would have differed by a factor of 5½, 2,400 to 13,224 I/O's. Figure 6 shows the effect of the compression factor on hybrid hash join performance for inputs R and S. The numbers above the solid curve indicate the exact I/O cost of hybrid hash join for the compression factors marked at the bottom axis. The dashed line indicates the I/O count for the inputs only; the difference between the solid and dashed lines is the

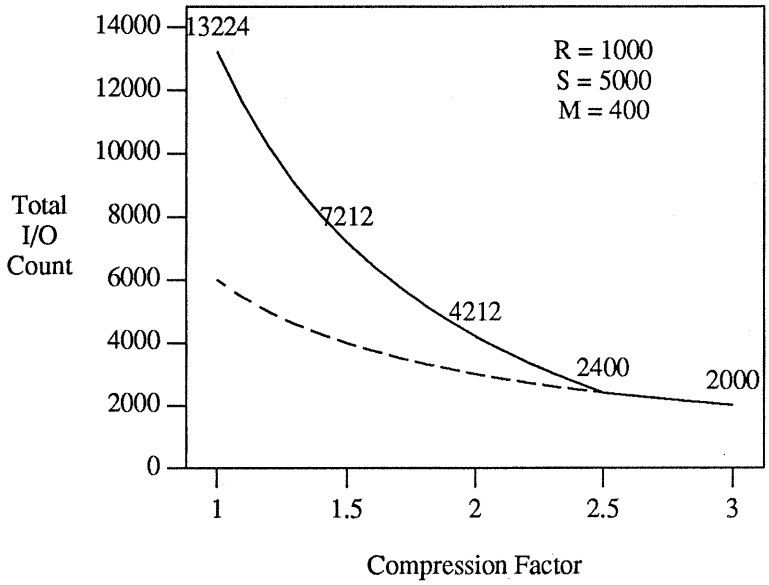


Figure 6. Effect of Compression on Join Performance.

I/O cost for the partition files. In other words, if the inputs are pipelined into the join operation, the distance between the dashed and the solid lines represents the join's I/O cost. In the best case with good compression factors, this cost can be zero.

The graph can be divided into two regions. For compression factors below 2½, the build input is larger than memory, hash table overflow occurs, and I/O reduction by compression is more than the compression factor, similar to the example above. For compression factors above 2½, no overflow occurs and compression only reduces I/O on permanent files. However, it is important to observe in this graph that already very moderate compression factors, e.g., 1½, reduce the total I/O cost significantly. Even if some additional cost is incurred for decompressing output data, which is probably a very small amount of data compared to the data volumes involved in the first query processing steps, the performance gain through compressed permanent and temporary data on disk and in memory far outweights the costs of decompression.

Figure 7 shows the effect of compression on hybrid hash join performance for a variety of memory sizes. The bottom-most curve for a memory size of 1,000 pages reflects the situation without overflow. The curve for 500 pages of memory has a steep gradient up to compression factor 2. Beyond this point, the hash table fits into memory and the curves for 500 and 1,000 pages coincide. For 250 pages of memory, which is ¼ of R, the curve joins the other curves without overflow at a compression factor of 4. For all smaller memory sizes, the hash table does not fit into memory in the considered spectrum of compression factors. However, the performance gain is more than the compression factor for all memory sizes. For 50 or 100 pages of memory, the curves are very close to each other, because almost all of R and S must be written to overflow files. If memory is very limited in a system, it might be more important to allocate it to operators that may be able to run without overflow, and to use memory there with maximal efficiency, i.e., the best compression scheme possible.

Figure 8 shows the speedup for the previous figure. The bottom-most curve, for 1,000 pages of memory, represents linear speedup. All other curves indicate super-linear speedup. The curve for 500 pages has an obvious "knee" at compression factor 2, which had been already visible in the previous figure. For 334 pages of memory, the knee would be located at compression factor 3, at the edge of the graph, where the curve indicates a speedup factor of 7. Considering that a speedup of 7 could be achieved with a compression factor of only 3 makes it imperative to exploit compression for database query processing performance, independently of whether or not disk space savings provide an additional incentive to use compression.

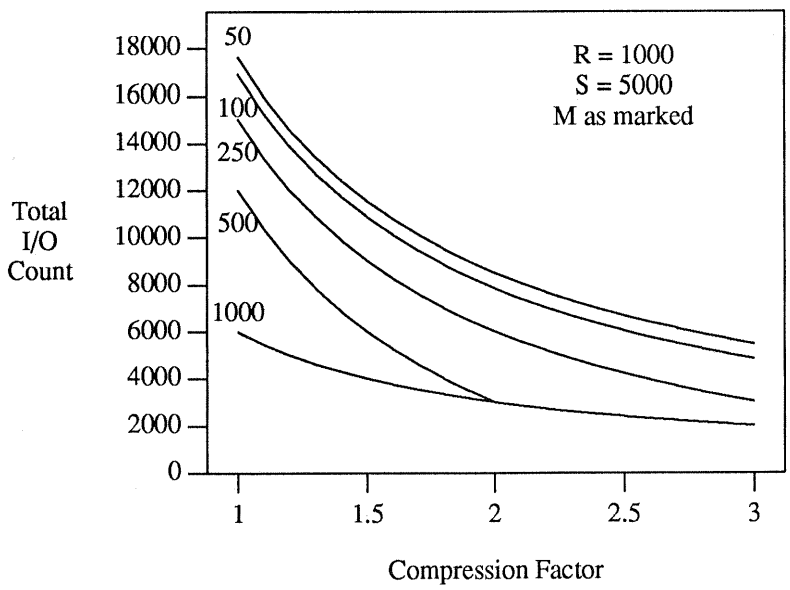


Figure 7. Effects of Compression and Memory Size.

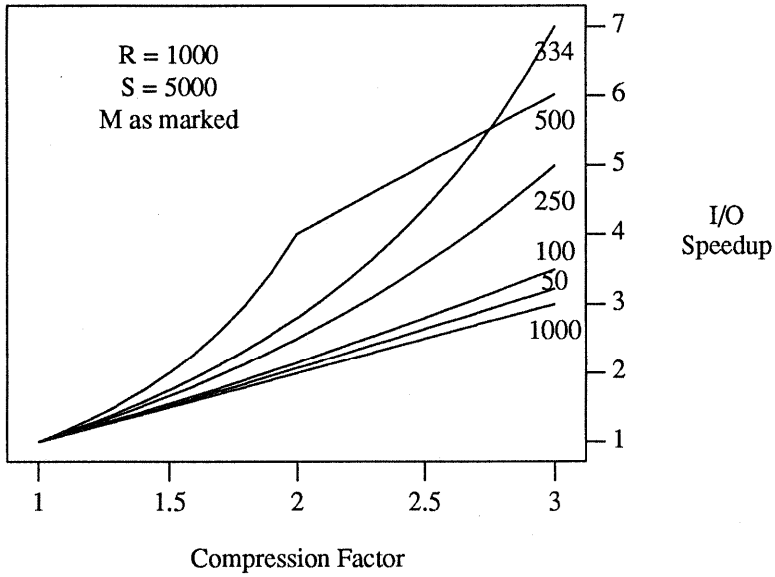


Figure 8. Speedup of Hybrid Hash Join through Compression.

Summary of Compression for Query Processing

In summary, query processing using compressed data has two advantages. First and obviously, I/O to permanent and temporary files is faster when measured as information content (records) per time unit; in fact, all data transfer between levels of the memory hierarchy is faster for compressed than for uncompressed data. Second, all levels in the memory hierarchy seem larger; this has particular useful effects on buffer hit rates, index fan-out, and working space in memory for hybrid hash join and similar matching operations. The first advantage translates into a linear speedup — a compression factor Z makes query processing Z times faster — while the second advantage permits super-linear speedup. The requirement for obtaining these advantages is that fixed compression schemes be used for each domain, easily accomplished by using ADT mechanisms available in modern database management systems. Dynamic, adaptive compression schemes can be employed by performing the adaptive component of the compression algorithm during database reorganization and then leaving the compression scheme unchanged during database operation.

Finally, considering the improvement rates of CPU and disk speeds, compression may become a viable means for overflow resolution in hash-based query processing algorithms. It might be faster to compress data than to write them to overflow files and read them back into memory. We leave this possibility for future analysis.

4. Fan-out and Recursion Depth

Hash-based query processing algorithms rely on partitioning to divide large inputs into manageable fragments, possibly using multiple recursion levels. The major cost of partitioning is the I/O cost; therefore, making I/O as fast as possible is very important.

Improving I/O Throughput

There are several ways to speed up the I/O during partitioning. First, multiple disk drives can be employed, effectively increasing not only the I/O bandwidth but also the number of seeks that can be performed per unit of time. In the extreme case, if the number of available disks is equal to the partitioning fan-out, disk seeks may be virtually eliminated. This configuration might become effective in the near future with very fast processors and relatively slow disks. For the use of redundant arrays of inexpensive disks (RAID's) [30] in database query processing, this might mean that RAID disk controllers that provide the abstraction of one large, reliable disk by hiding the individual disk drives from the database management system do not permit obtaining the highest query processing performance.

Second, disk space allocation for partition files might be tuned to ensure very fast writing during partitioning, at the expense of slower reading while processing individual partition files. For example, pages of partition files may be written to consecutive disk locations without regard to which of the partition files a page belongs to. The design of a disk space allocation strategy that balances write and read speed is an open issue. A solution would improve the performance of both partitioning and external sorting, because partitioning and merging are dual processes [18].

Third, I/O is faster with larger clusters, i.e., larger units of I/O. While the transfer time is constant for a given data volume, the number of disk accesses and therefore the total time for disk seeks and rotational latencies is smaller for larger clusters. This is exploited, for example, in the design of many disk caches. However, we assume the absence of disk caches for now and consider their effect towards the end of this section. With larger cluster sizes for the partition files, each individual recursion level is faster, but unfortunately the partitioning fan-out is reduced and the number of required recursion levels may therefore be increased. In other words, increasing the cluster size has two opposite effects on hybrid hash join performance. Determining the optimal cluster size must therefore consider both performance effects, and depends mostly on the ratio of disk transfer speed and access time (seek plus rotational latency). In this section, we explore the effect of larger cluster sizes on the performance of recursive hybrid hash join.

Performance Observations

The time spent writing and reading partition files in recursive hybrid hash join can be approximated by

$$2 \times \log_{M/C}(R/M) \times (R+S)/C \times (A + C \times X)$$

$$= \left[2 \times \ln(R/M) \times (R+S)\right] \times \left[(A + C \times X)/\ln(M/C)/C\right]$$

for input size R and S, memory size M, cluster size C, disk access time A (seek plus rotational latency), and transfer time X for one page. R is the build input and determines the recursion depth. This formula does not include rounding and is therefore only an approximation. In the second line of the equation, terms constant in C are moved into the first term and terms that influence the optimal choice of C are in the second term. The first interesting observation of this formulation is that the input sizes have no impact on the optimal choice of C, which depends only on memory size and disk performance parameters. Thus, in a pipelined environment in which the input sizes can only be roughly estimated by the query optimizer, the cluster size can be chosen before a hybrid hash join actually starts.

For example, consider a hybrid hash join with $R = 40{,}000$ pages, $S = 50{,}000$ pages, $M = 200$ pages, $A = 25$ msec, and a transfer speed of 2 MB/sec or $X = 2$ msec per 4 KB page. For the minimal cluster size ($C = 1 = 4$ KB), a single partitioning level suffices and the I/O time is $2 \times \log_{200}(200) \times 90{,}000 / 1 \times 27$ msec $= 81$ min. For fairly large clusters of $C = 16 = 64$ KB, more than two recursion levels are required and the I/O time is $2 \times \log_{12}(200) \times 90{,}000 / 16 \times 57$ msec $= 22.8$ min, or 3½ times faster than hybrid hash join with minimal cluster size and recursion depth. This example demonstrates that large clusters and deep recursion are much more efficient than a cluster size that permits minimal recursion depth, at least for large inputs for which the approximation formula above is reasonably accurate.

Figure 9 shows, for a variety of memory sizes, the optimal choice of cluster sizes and the resulting maximal build input size that can be handled by hybrid hash join, i.e., without recursion. The latter value was calculated as $M \times F = M \times M / C$. Note that these values were obtained by using the approximate hybrid hash join formula without rounding; however, since more detailed experiments have shown that the performance is fairly stable for cluster sizes near the optimal cluster size and since the disk performance characteristics of real disks vary around the values we assumed, this figure illustrates our argument about the optimal cluster size and maximal build input size with sufficient accuracy. For example, Figure 9 indicates that for a memory allocation of 512 KB, the optimal cluster size is 60 KB. Thus, the maximal fan-out is 8 and the maximal build input size for hybrid hash join is 4 MB. Clearly, for mainframe computers in which a memory allocation of 512 KB to a single operator of a single query may be reasonable, inputs above 4 MB will occur frequently. Furthermore, if a large number of operators are active concurrently in a right-deep or bushy query evaluation plan [32], the memory allocation for each operator might be fairly small. In order to process larger inputs, multiple recursion levels are required.

Figure 10 shows the I/O times for partition files of hybrid hash join with a variety of cluster sizes for $R = 10{,}000$ pages, $S = 50{,}000$ pages, $M = 200$ pages, $A = 25$ msec, and $X = 2$ msec per page. It is immediately apparent that small clusters and large fan-outs result in disastrous performance. The best performance, with 25-

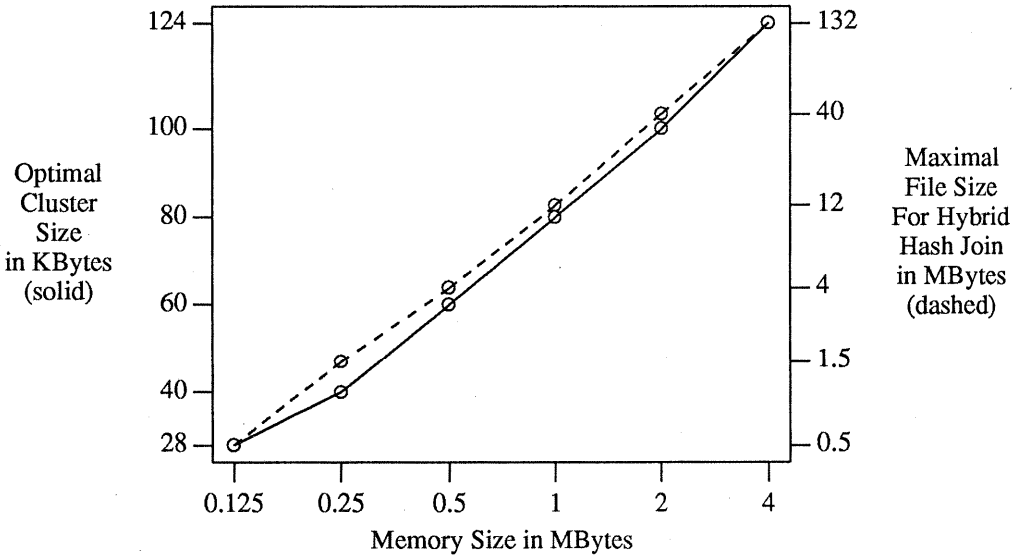


Figure 9. Optimal Cluster Sizes by Memory Size.

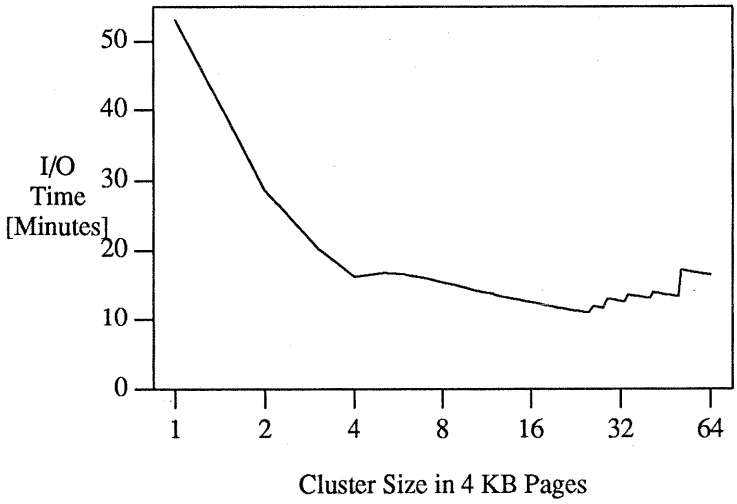


Figure 10. Performance Effect of Cluster Size Choice.

page clusters, is 4½ times better than the performance for single-page clusters. The sawtooth pattern for larger cluster sizes is due to rounding the fan-out in the cost formula.

Disk Caches

Many modern disk drives attempt to use "track-at-a-crack" I/O even if the host computer issues I/O requests at the granularity of pages. Nonetheless, we did not take disk caches into consideration in the above discussion. We believe that two reasons justify this omission. First, most disk drives do not cache writes, only reads, for reliability reasons and because cache organization is less complex. Such disk caches would have no effect on partitioning. Second, if the disk drive does cache writes but the number of "cache lines" (tracks) in the disk cache is smaller than the number of clusters in the operator's work space available for output buffers (M / C is our discussion above), the disk cache will thrash and will not enhance performance. However, as disk caches grow from one generation of disk drives to the next, this question will require reexamination, in particular if it turns out that disk caches grow faster than main memories and typical memory allocations to query execution algorithms.

Summary of Cluster Size and Fan-out Tuning

To summarize this section, we conclude that (i) the I/O characteristics of real disk drives require fairly large clusters for optimal performance, (ii) hybrid hash join can be several times faster with large clusters than with small clusters, (iii) the optimal partitioning fan-out for hybrid hash join is quite small, and (iv) multiple recursion levels are required for realistic input sizes. In earlier research into the performance of external sorting, a similar tradeoff based on cluster sizes had been observed in measurements of a working system, namely larger merge fan-in and fewer merge levels with small clusters vs. faster I/O with large clusters [15]. The conclusion, as in the case of hash partitioning, had been that relatively large clusters and a relatively small fan-in give the best performance, which is not surprising considering the duality of merging and partitioning [18]. In the next two sections, we use successive recursion levels to manage and even exploit unpredictable inputs sizes as well as data and hash value skew.

5. Role Reversal

In complex queries, it is impossible to estimate sizes of intermediate results and the final result reliably and accurately [7, 20]. For example, if a join input is the result of a selection, another join, or even a complex subquery, the estimation error for the intermediate result size can easily be one or two orders of magnitude. Therefore, it is frequently impossible to decide which of the two inputs of binary operation is smaller than the other and should be the build input for hybrid hash join.

Figure 11 shows the cost of merge-join and hybrid hash join for a wide range of relative input sizes, given a fixed sum of the two input sizes of 50,000 pages, 100 pages of memory, and a fan-out of 10. It is apparent that the cost of merge-join is relatively stable across the entire range in Figure 11. The reason is that the merge depth of sorting is determined for each input individually, and since the size of the larger input is at least ½ of the sum of sizes, the number of times the majority of records is written to and read from temporary files is about constant across the entire range.

On the other hand, hybrid hash join shows dramatic performance benefits if the build input is significantly smaller than the probe input. This is due to the fact that the recursion depth is determined only by the build input. In the extreme case, when $\log_2(R / S) = -9$ and $R = 50000 / (2^9 + 1) < 100 = M$, the build input fits into memory and no I/O to or from temporary files is required. If the build input is the larger of the two inputs, the performance of hybrid hash join and merge-join is very similar. Thus, ensuring that the build input is always chosen to be the smaller input is very important. Since query optimizers cannot reliably predicate relative sizes, this choice must be delayed until run-time. After a partitioning step has been completed, the hybrid hash join algorithm can use the

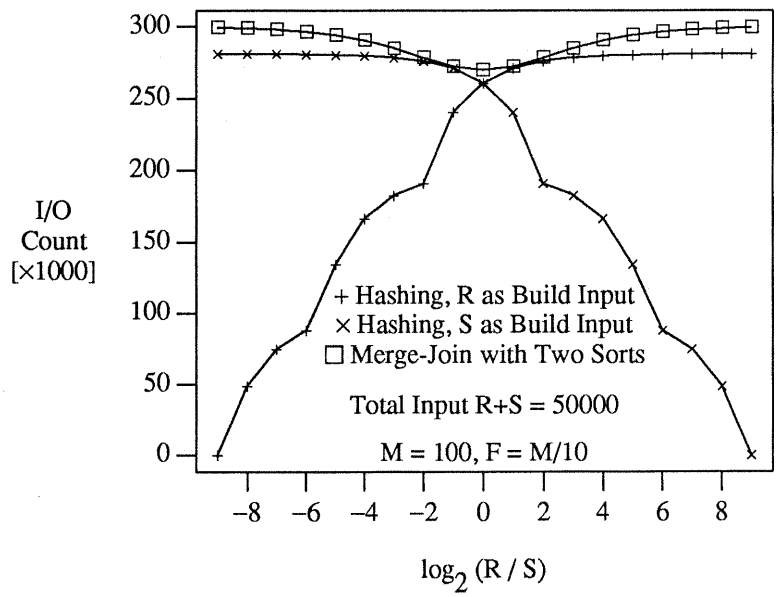


Figure 11. Merge-Join and Hybrid Hash Join Costs.

sizes of the files just written to determine whether to reverse the roles of build and probe inputs. This is a conceptually straightforward optimization of hybrid hash join, and some implementations of hybrid hash join already exploit role reversal.

With dynamic role reversal, the smaller of the two inputs determines the recursion depth, and the recursion depth for joining R and S using memory M and partitioning fan-out F for hybrid hash join with the ability to reverse build and probe roles is

$$\min\left(\log_F (R / M), \log_F (S / M) \right) = \log_F \left(\min(R, S) / M \right).$$

However, this upper bound is not very tight, because it is reestablished at each recursion level for each partition. In other words, for almost equal input sizes, the assignment of original inputs to build and probe roles are set individually and may be different for each partition. Thus, facilities for role reversal not only after the first partitioning step but for each partition at each recursion level are most effective in those joins in which the performance difference between merge-join and hybrid hash join is minimal.

6. Non-Uniform Hash Value Distributions

The efficiency of set processing algorithms based on partitioning depends on the effectiveness of the partitioning scheme. If the partitioning scheme cannot ensure that the partitions are of equal or at least similar size, performance will be poor. If the hash value distribution and the partitioning scheme result in 99% of all input items being assigned to a single partition, one recursion level is wasted, and the recursion depth of the entire operation will be larger than $\log_F (R)$ for fan-out F and build input R.

Earlier Skew Management Schemes

In order to deal with data value skew (non-uniform distributions), the Grace database machine uses bucket tuning and dynamic destaging [23, 24, 28]. The basic idea is to create many more partitions than anticipated to be necessary and to "destage" the largest resident build partition from memory to disk whenever overflow occurs.

This method is based on three assumptions. First, because destaging decisions are based on information about the build input only, dynamic destaging deals effectively only with skew in the build input. If the hash value distribution of the build input is uniform but the hash value distribution of the probe input is highly skewed, dynamic destaging misses significant optimization opportunities. Second, the dynamic destaging algorithm decides which partition (or bucket) to destage to disk (i.e., write to a partition file) based on build input items inspected at the point overflow occurs. For a build input 100 times the size of memory, the first destaging decision is based on only the first 1% of the build input items. This is not a random sample of the input; therefore, if the build input is not in random order with respect to the join key, a destaging decision might well be wrong. Third, in order to create the opportunity for choice, many more partitions must be created than truly required for the present build input size. Thus, the fan-out must be chosen artificially large, with the detrimental effects on cluster size and I/O performance discussed earlier.

For the Gamma database machine, Schneider investigated the effects of skew on the performance of hybrid hash join, considering both load imbalances in parallel systems and the effects of skew on local (sequential) algorithms. For moderate skews, he observed limited effects on the performance of hybrid hash join, but recommended sorting and merge-join for data with strong skew [32]. More recent investigations of DeWitt, Naughton, Schneider, and Seshadri have focused on the effect of skew on parallel execution algorithms and on load balancing, and have proposed a number of techniques based on sampling permanent database files or materialized intermediate query results [12, 13, 34]. Similarly, Walton et al. have classified a number of skew types and considered their effect on the speedup behavior of parallel joins [37]. In contrast to this work, we focus here on the effects of skew in sequential hybrid hash join, and develop techniques that are effective in both sequential and parallel execution environments.

Algorithm Motivation and Intuition

For the one-to-one match match operator in the Volcano query execution engine [17, 21], we designed and simulated a new skew management scheme that actually exploits skewed data and hash value distributions for improved performance [5]. Because the algorithm is a version of hybrid hash join modified to capture hash value distributions when partitioning large inputs recursively, we called it *histogram-driven recursive hybrid hash join.* Its basic premise is that multiple recursion levels are used for large inputs, i.e., the case in which skew has the

largest effect. The most important modification is that histograms are gathered while writing a partition file, saved with the file until it is processed, and used when the partition file is partitioned in the next recursion level. Other forms of statistics could also be used, e.g., variance and covariance of the bits in all hash values.

Our simulations have indicated that fairly simple statistics are quite effective. When creating a partition file, the algorithm also initializes a histogram with H counters, where $2 \times F \leq H \leq 5 \times F$ is sufficient. For the memory sizes and fan-outs considered in the previous section, this requires that less than 1% of memory be used for histograms. Note that there are H counters for each output file; thus, while partitioning with fan-out F, a total of $F \times H$ counters are maintained. For each item written to the file, a counter in the histogram is incremented by the size of the item. When the file is closed, the histogram is appended to the file and will be available when the file is reopened for reading. The histogram is destroyed at the same time as the file.

The purpose of the histogram associated with a partition file is to provide guidance for partitioning this file in the next recursion level. Therefore, the counter to be incremented for each item is determined using the hash function that will be employed in the next recursion level. Using this hash function, the partition being written is interpreted as consisting of H subpartitions, which we call *virtual subpartitions*, because H virtual subpartitions are written to a single partition file. However, when this file is partitioned in the next recursion level, the histogram provides advance knowledge how the H virtual subpartitions should be assigned to an in-memory hash table and up to F disk-based subpartition files.

First Partitioning Step Without Histograms

The algorithm first chooses the optimal fan-out F and then partitions the build input into F partition files. The original build and probe inputs are obtained from two input plans that may consist of a single file scan or of many complex operations. Considering the errors inherent in database selectivity estimation [7, 20], sizes and value distributions of the inputs are presumed to be entirely unknown.

The first partition step, which presumes no information about sizes and data and hash value distributions in its inputs, is quite similar to bucket tuning [23, 28]. The important difference is that our first partitioning step uses the standard fan-out F, which is relatively small, in order to achieve high I/O throughput, whereas bucket tuning depends on a large number of output partitions with small cluster sizes to create possibilities to combine buckets. At the end of the first build phase, the information gathered on the build input is used to execute hybrid hash join. Our understanding and implementation of hybrid hash join includes the possibilities that in-memory hash join is feasible ($R \leq M$) or that none of the build partitions is smaller than memory, i.e., no in-memory hash table can be used. First, the build partitions are sorted by their size. Call the sizes of build partitions by increasing size R_i. Second, the smallest $\bar{K}$ build partitions are assigned to an in-memory hash table. The remaining $K = F - \bar{K}$ partitions must be written to disk. $\bar{K}$ and K are determined to ensure that the in-memory partitions leave sufficient memory for K output buffers. Thus, $\bar{K}$ is assigned the largest value that does not violate

$$\sum_{i=1}^{\bar{K}} R_i \leq M - K \times C = M - (F - \bar{K}) \times C.$$

The largest of these partitions might be larger than one cluster; in that case, the $\bar{K}$ partially filled output buffers of the partitions assigned to the in-memory hash table are compacted to eliminate fragmentation and to create free space for the clusters that were already written to disk and must be read back into memory. For $\bar{K} = F$ and $K = 0$, this hybrid hash join is effectively in-memory hash join. If $\bar{K} = 0$ and $K = F$, all output buffers are used for partition files. Notice that even if $R > F \times M$, an in-memory hash table is used if the smallest build partition is smaller than one cluster. Furthermore, if the memory size is not a multiple of the cluster size ($M > F \times C$), the remaining memory is used as part of the hash table.

This assignment of partitions to memory and to disk is then used to partition the probe input and to perform the join for partitions assigned to memory. After the first level of partitioning is complete, there are K pairs of partition files left to be joined. These joins are different from the original join, because the sizes of the join inputs (which are partition files) and their virtual subpartitions are known from the histograms associated with the partition files. Before reading, partitioning, and joining a pair of partition files, histogram-driven recursive hybrid hash join uses the information on virtual subpartitions to assign virtual subpartitions to an in-memory hash table and to partition files to be joined in the next recursion level. Table 1 shows an example histogram for 8 virtual subpartitions, including size indicators for the build and probe virtual subpartitions and classifications that will be discussed in the next subsection.

i	r_i	s_i	classification
0	5	6	normal
1	0	4	empty
2	2	18	normal, in memory?
3	8	13	oversized
4	3	0	empty
5	4	1	normal, role reversal?
6	1	5	normal
7	12	17	oversized

Table 1. Example Histogram of Build and Probe Virtual Subpartitions.

Histogram-Driven Recursive Hybrid Hash Join

There are a number of goals that can be pursued and must be balanced against one another using the information on virtual subpartitions. These goals include assigning virtual subpartitions to memory or to partition files or to decide against partitioning and choose an alternative algorithm. The latter choice is particularly useful when very high numbers of duplicate data values render partitioning useless. The common thread in all these goals is to save I/O.

The histogram-driven recursive hybrid hash join algorithm proceeds in several steps at each recursion level. First, role reversal is considered if the probe input is smaller than the build input. Therefore, we assume in the sequel that the build input file is not larger than the probe input file. We will continue to call the build input file R and the probe input S, even if the roles of R and S have been reversed.

Second, if in-memory hash join is feasible ($R \leq M$), it is used and the subsequent steps do not apply. This case will eventually be reached after some number of recursion levels and will terminate the recursion.

Third, the virtual subpartitions are classified according to their histogram entries r_i and s_i indicating the sizes of the virtual subpartitions in build and probe input. The virtual subpartitions are classified as belonging into one of three groups, namely as *empty* if the build input or the probe input do not contain items belonging to this virtual subpartition ($r_i = 0$ or $s_i = 0$, such as entries 1 and 4 in Table 1), *oversized* if the build input is larger than memory ($r_i > M$, such as entries 3 and 7 in Table 1 presuming $M = 6$), or *normal* for the remaining virtual subpartitions. Empty virtual subpartitions are ignored; if the operation is not a join but an outer join, anti-semi-join, union, or difference, items belonging to empty virtual subpartitions can be transformed into output without the use of a hash table or partitioning. In a sense, empty virtual subpartitions are used like symmetric bit vector filtering with H bits for each of build and probe input, which we call the filter effect of histogram-driven recursive hybrid hash join. Oversized virtual subpartitions form separate output partitions during hybrid hash join; each oversized virtual subpartition is assigned to its own partition file. If the number of oversized virtual subpartitions is called Z and $Z > F$, hybrid hash join does not apply and partitioning with full fan-out F is used as described below in the sixth step of histogram-driven recursive hybrid hash join. If $Z \leq F$, hybrid hash join is planned for the normal virtual subpartitions using memory equal to $M - Z \times C$. If the number of normal virtual subpartitions is very small, alternative join methods such as nested loops are considered, as discussed below as the eighth step of histogram-driven recursive hybrid hash join.

Fourth, the normal and oversized virtual subpartitions are sorted by the size in the build input (r_i) and by the quotient $Q_i = r_i / (r_i + s_i)$. For the example in Table 1, the sort order by size is $i =$ 6, 2, 5, 0, 3, 7; by quotient Q_i, it is $i =$ 2, 6, 3, 7, 0, 5. Although oversized virtual subpartitions form individual output partitions and therefore do not participate in the fifth step of histogram-driven recursive hybrid hash join, i.e., the crucial planning step, oversized virtual subpartitions are included in the two sorts because some oversized virtual subpartition might be re-classified to normal later in the sixth step. The quotient Q_i expresses the relative cost and benefit of keeping a virtual subpartition in memory vs. assigning it to a partition file. If a virtual subpartition i is assigned to memory, the required space in the hash table is given in r_i. If a virtual subpartition is assigned to a partition file, the I/O cost will be proportional to $r_i + s_i$. Therefore, the quotient Q_i captures the importance of assigning virtual subpartition i to the in-memory hash table. For example, keeping virtual subpartition 6 in memory creates savings proportional to $r_6 + s_6 = 6$, while keeping virtual subpartition 2 in memory creates savings proportional to $r_2 + s_2 = 20$, more than twice than of virtual subpartition 6. This is captured in $Q_2 = 2 / 20 < Q_6 = 1 / 6$. Note that an algorithm that only considers build sizes (equivalent to the r_i in Table 1) would have prefered virtual subpartition 6 over virtual

subpartition 2.

Fifth, the normal virtual subpartitions are assigned to memory or to a partition file. Calling the sum of the build sizes of the normal virtual subpartitions $\bar{R}$ ($\bar{R} = 5 + 2 + 4 + 1 = 12$ in Table 1), the number of partitions K required for the normal virtual subpartitions is calculated to satisfy $K \times M + (M - (K + Z) \times C) \geq \bar{R}$. Thus, $K = \left\lceil (\bar{R} - (M - Z \times C)) / (M - C) \right\rceil$. If $K > F - Z$, hybrid hash join does not apply and the algorithm skips directly to its sixth step. Otherwise, the size of the in-memory hash table is set to $M - (K + Z) \times C$, which may be empty if $K = F - Z$. For most effective use of this memory space, virtual subpartitions are fit into the in-memory hash table in the order of increasing quotients Q_i until the entire hash table is filled. The remaining virtual subpartitions are assigned to K partition files using a bin packing algorithm such that no build partition file is larger than M. Since precise bin packing is NP-complete, we use a decreasing (by build size) first fit heuristic, which performs well for this application.

If the bin packing algorithm fails, i.e., no assignment of all normal virtual subpartitions to the in-memory hash table and to output partitions no larger than memory can be found, the number of output partitions K is incremented by 1 (with the corresponding decrease in hash table size) and a new assignment is attempted. In rare cases, K may need to be incremented repeatedly until a feasible assignment is found or K would exceed $F - Z$. Once a feasible assignment is found, hybrid hash join is executed with K output partitions using the determined assignment of virtual subpartitions to memory and to partitions. If $K = F - Z$ and no feasible assignment can be found, hybrid hash join does not apply, and full partitioning is used as described below as the sixth step of histogram-driven recursive hybrid hash join.

If the build input is greater than the usual limit for hybrid hash join ($\bar{R} > F \times M$), hybrid hash join will typically not apply unless there are empty virtual subpartitions or several very large oversized virtual subpartitions that make hybrid hash join for the remaining normal virtual subpartitions feasible. However, since these cases can occur and make hybrid hash join usable even when $\bar{R} > F \times M$, the histogram-driven recursive hybrid hash join algorithm always attempts to plan hybrid hash join before resorting to partitioning with full fanout F.

If the expected size relationship of build and probe inputs does not hold for some virtual subpartition i, i.e., $r_i > s_i$ or $Q_i > ½$ (such as entry 5 in Table 1), it is possible to reverse their roles. However, role reversal for individual virtual subpartitions works only for virtual subpartitions assigned to partition files, not those assigned to memory. This is not a strong requirement, because the quotient Q_i is particularly low for virtual subpartitions assigned to memory while Q_i is particularly high for virtual subpartitions that might benefit from role reversal. A stricter requirement is that the sets of virtual subpartitions with and without role reversal must be assigned to different partition files to limit the number of necessary output buffers during partitioning. In other words, either all or none of the virtual subpartitions assigned to one partition file must use role reversal. In order to optimize the total I/O costs, it might be useful to reverse roles for some virtual subpartitions for which $Q_i \leq ½$.

The assignment of virtual subpartitions based on their quotient Q_i is the crucial element that permits the histogram-driven recursive hybrid hash join algorithm to not only manage but even exploit hash value skew in the inputs. By choosing those virtual subpartitions to remain in memory that permit the greatest I/O savings with the least amount of memory, this algorithm executes with skewed input data more efficiently than with perfectly uniformly distributed data. The amount of I/O for the build input is necessarily $\bar{R} - (M - K \times C)$ for a build input partition file R. If all values appear with uniform frequency in both inputs, the amount of I/O required for probe overflow files is $S \times (\bar{R} - (M - K \times C)) / \bar{R}$, i.e., the same fraction of S is written to disk as for R. However, by choosing the virtual subpartitions i with the smallest Q_i to remain in memory, the I/O for probe overflow files is guaranteed to be less than this amount. It is important to note that skew in either one or both of R and S will create this effect, because the sizes of virtual subpartitions from both inputs influence the quotient Q_i and permit "investing" memory for the greatest I/O savings.

Sixth, if a feasible assignment of all virtual subpartitions to F partition files smaller than memory could not be found and therefore hybrid hash join does not apply, the build and probe inputs are partitioned into F output partitions such that the build output files are of equal or very similar size. The goal is to reduce build input sizes as effectively as possible in order to perform the entire join with minimal recursion depth, which is achieved by grouping the virtual subpartitions according to the r_i counters. The sizes of probe virtual subpartitions, i.e., the s_i's, are not used in this step. The distribution of virtual subpartitions into partitions uses the bin packing algorithm with a bin size limit equal to the sum of build items in normal virtual subpartitions divided by the number of output buffer available for normal virtual subpartitions, i.e., $\bar{R} / (F - Z)$. The bin size limit is set such that the bin packing algorithm succeeds if it finds a perfect fit and distributes the normal build input into $F - Z$ partitions sized

as evenly as possible, i.e., the entire build input into F partitions. If the bin packing algorithm fails, the limit is increased by a small fraction (e.g., 5%) and a new assignment is attempted. If necessary, the limit is increased repeatedly until a fit can be found. For each new limit, oversized virtual subpartitions with less build input than the new limit are re-classified from oversized to normal, thus increasing the number of bins available to the bin packing algorithm for normal virtual subpartitions and therefore its freedom for assigning virtual subpartitions to partition files.

Seventh, as for virtual subpartitions and partition files during hybrid hash join, role reversal may be considered for individual virtual subpartitions when partitioning with full fan-out F and no in-memory hash table. Of course, the same restriction applies as for hybrid hash join, namely that all virtual subpartitions assigned to one output partition must either all use role reversal or none of them does.

Eighth, after an assignment of virtual subpartitions to partition files has been found, the expected cost of performing the partitioning step just planned and joining the resulting partition files using histogram-driven recursive hybrid hash join is compared to the estimated cost of using nested loops join instead, i.e., replacing the partitioning step and subsequent hybrid hash join with nested loops. If the number of duplicates is very high and partitioning and hybrid hash join are not a very effective join method, an alternative algorithm is chosen before an ineffective partitioning step is attempted, not after a partitioning step turned out to be ineffective. All partitioning steps that actually are performed are judged more effective than an alternative nested loops join.

Ninth, if hybrid hash join has been chosen and the bin packing algorithm succeeded in planning partition files no larger than memory, gathering histograms is disabled and bit vector filtering is used instead, because histograms will not serve any purpose if the next recursion level is known to use in-memory hash join only. Using bit vector filter filtering in the last recursion level is particularly effective, because the previous partitioning steps have reduced the number of distinct values that actually occur in a build partition file.

Performance Observations

In order to verify the effectiveness of histogram-driven recursive hybrid hash join, we simulated the algorithm as described above and compared it with naive hybrid hash join and with hybrid hash join using bit vector filtering in each recursion level. Naive hybrid hash join is hybrid hash join without any skew management techniques, i.e., all recursion levels use the algorithm described above for the first partitioning step. The size of the bit vector filters was set equal to the number of bits in the counters used in histogram-driven recursive hybrid hash join, assuming 32 bits per counter. For example, histograms with 40 virtual subpartitions per output partition are compared with bit vector filtering with 1,280 bits per output partition. We also included an algorithm that uses both histograms and bit vector filters. In order to contrast algorithms with the same amount of memory for bit vector filters and histogram data, we compared histogram-driven recursive hybrid hash join with 40 virtual subpartitions and no bit vector filters with histogram-driven recursive hybrid hash join with 20 virtual subpartitions and 640 bits per output partition. All algorithms consider role reversal of build and probe inputs except in the initial partitioning step when the input sizes are presumed to be unknown.

In the following diagrams, we varied one or two parameters and left all other parameters at the defaults shown in Table 2. The simulation uses integer join keys instead of entire records and in-memory arrays instead of files. The performance measure is the number of record I/Os relative to the number of I/Os estimated by the cost formulas given in the overview of hybrid hash join. The input values were chosen using the UNIX random() library function, which produces a uniform random distribution. For tests with skewed inputs, we changed the

R	Cardinality of Build Input	10000
S	Cardinality of Probe Input	20000
D	Cardinality of Domain	10000
RZ	Skew in Build Input	0
SZ	Skew in Probe Input	0
M	Memory Size (Records)	800
F	Fan-out	10
H	Histogram Size (per output partition)	40
B	Bit Vector Size (bits per output partition)	1280

Table 2. Parameters and Defaults in Skew Experiments.

random function used to generate uniform input data to take a uniformly random value U chosen from [0,1) to a power z before multiplying it with the domain size D, where z is the measure of skew. For $z \geq 0$, the skewed value is calculated as $\lfloor D \times U^{1+z} \rfloor$; for $z < 0$, as $\left\lfloor D \times \left\lceil 1 - U^{1-z} \right\rceil \right\rfloor$. Each data point in the following diagrams is the average of 10 runs.

Figure 12 shows the I/O counts for uniformly distributed input data for various numbers of virtual subpartitions maintained for each output partition. The data in Figure 12 show the effectiveness of the hybrid hash join variants for data with only the little skew that is always found in random data. Zero virtual subpartitions indicate naive hybrid hash join without histograms or bit vector filtering.

Figure 12 suggests three interesting observations. First, the non-uniformity present in uniform random data is sufficient to make histogram-driven recursive hybrid hash join perform better than naive hybrid hash join. This suggests that good hash functions are no substitute for but complementary to histograms for managing and exploiting hash value skew. Second, both bit vector filtering and histograms improve the performance of hybrid hash join, increasingly so as more memory is added for the bit vector filter or the histogram. However, histograms appear to be much more effective than bit vector filters for small allocations of memory for statistics. Third, increasing histogram sizes does not result in linear performance improvements, while bit vector filters become more effective as their sizes increase. Not surprisingly, the combination of histograms and bit vector filters performs significantly better than either "pure" hybrid hash join variant. In a sense, the combined algorithms reaps the benefits of both methods. The performance effect from each method is slightly less than in either pure enhancement in Figure 12 because the amount of memory for statistical data was divided between histograms and bit vector filters.

Figure 13 shows the performance for hybrid hash join with bit vector filtering and histograms for skewed build inputs and uniformly distributed probe inputs. Not surprisingly, the diagram is symmetric due to the interpretation of negative skew in these experiments. The naive hybrid hash join algorithm is slowed down by moderate skew but improves with very strong skew because our implementation of naive hybrid hash join becomes quite effective in the second recursion level. Both join enhancements benefit from skew in the build input, although for very different reasons. The skew decreases the number of join values that actually occur in the build input, thus making bit vector filtering more effective. This is particularly visible for extreme skews. On the other hand, histogram-driven recursive hybrid hash join gains by determining which buckets have a particularly beneficial ratio of build and probe items (ratio Q_i in the discussion above) and by keeping these buckets in memory. Additionally, empty virtual subpartitions create the filter effect of histogram-driven recursive hybrid hash join. The combined algorithm is more efficient than either pure version, because it exploits the effects of both enhancements.

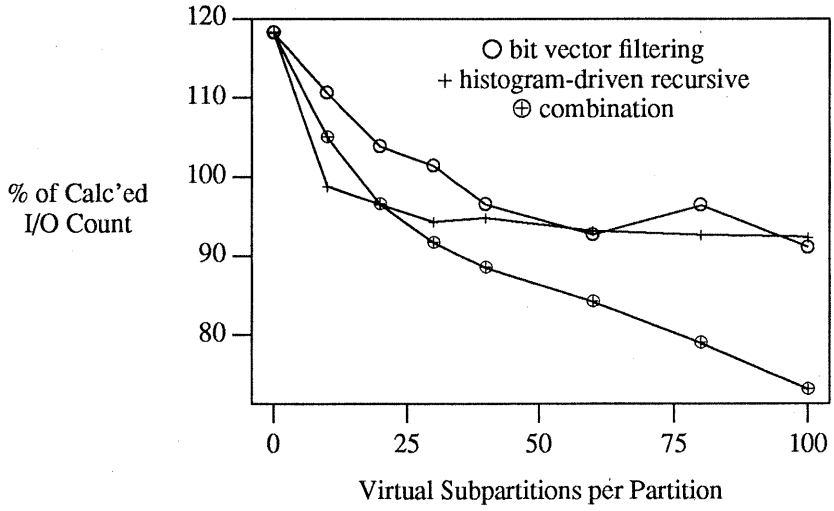


Figure 12. Effect of Histogram Sizes for Uniform Inputs.

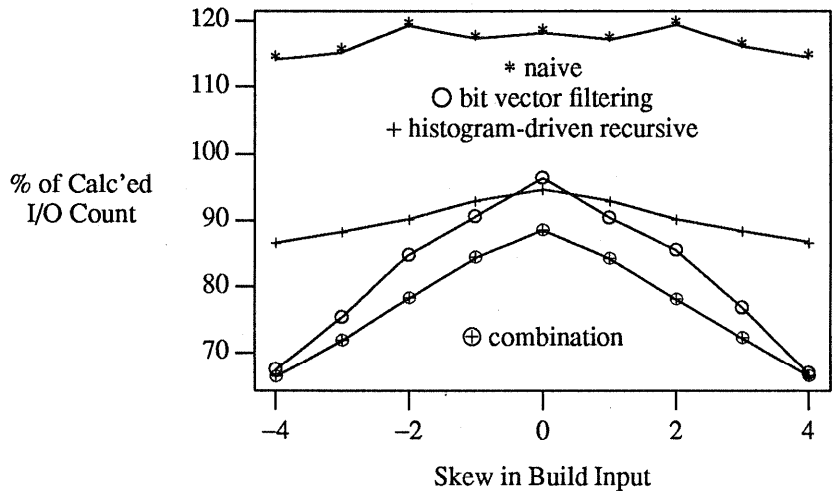


Figure 13. Performance for Skewed Build Inputs.

Figure 14 shows a similar experiment, but for skewed probe inputs and uniform build inputs. Naive hybrid hash join deteriorates under skew, as was to be expected and was observed in experiments comparing the effectiveness of sort- and hash-based query processing algorithms [18]. Since almost all possible keys actually do occur in the build inputs, bit vector filtering has no effect. Just as for naive hybrid hash join, the performance deteriorates with increasing skew. In contrast, histogram-driven recursive hybrid hash join makes informed assignments of buckets to memory and to partition files, because it considers bucket sizes of both build and probe inputs. It does not perform quite as well as with skewed build inputs, because of the filter effect of histogram-driven recursive hybrid hash join does not apply here. The small improvement from histogram-driven recursive hybrid hash join without bit vector filtering to the combined algorithm is quite interesting, because it shows that the controlled assignment of buckets to memory and overflow partitions increases the effectiveness of bit vector filtering. The important conclusion from Figure 14 is that histogram-driven recursive hybrid hash join exploits both build and probe skew, while bit vector filtering and naive hybrid hash join can only benefit from build skew and have no effect in the case of skewed probe inputs.

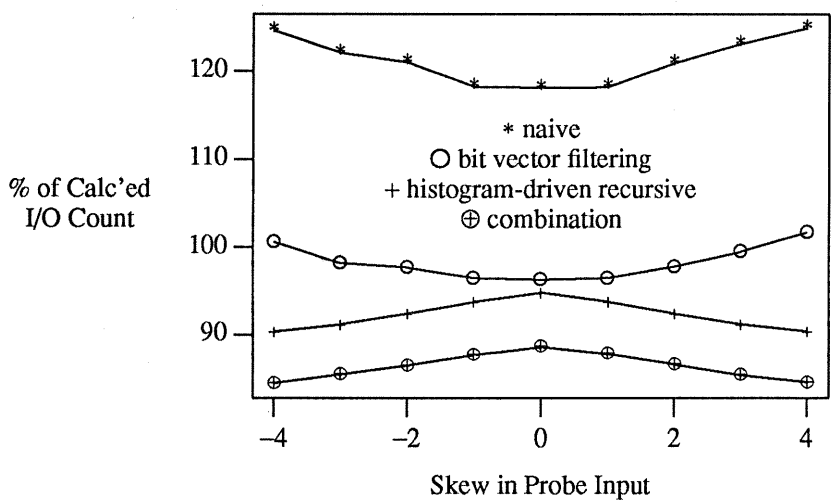


Figure 14. Performance for Skewed Probe Inputs.

Figure 15 shows the effect of two skewed inputs, with both inputs skewed similarly. In other words, the same values appear frequently in the build and in the probe inputs. Naive hybrid hash join is affected very strongly by this skew. The performance of both hybrid hash join enhancements also diminishes for strongly, similarly skewed inputs compared to non-skewed inputs. However, histogram-driven recursive hybrid hash join not only maintains its advantage over bit vector filtering but actually increases its advantage for stronger skews, i.e., it is less affected by concurrent skew than bit vector filtering. In particular, it is quite stable under moderate skew. As in the previous experiments, the combined algorithm is superior to either pure algorithm.

Figure 16 shows the effect of two inputs skewed in opposite directions. The skew control for the probe input is set to the negative value of the skew control for the build input. Thus, values that appear frequently in the build input are infrequent in the probe input and vice versa. The curve for naive hybrid hash join shows a similar shape as for skewed build inputs and uniform probe inputs. Small skews make the performance worse, while strong skews permit a more effective second recursion level. In contrast, bit vector filtering becomes very effective with increasing skew, because the skew decreases the number of values in the build input and, additionally, increases the number of probe items never written to partition files since they do not pass the bit vector filter. For very strong skews, bit vector filter is faster than histogram-driven recursive hybrid hash join. The positive effect of

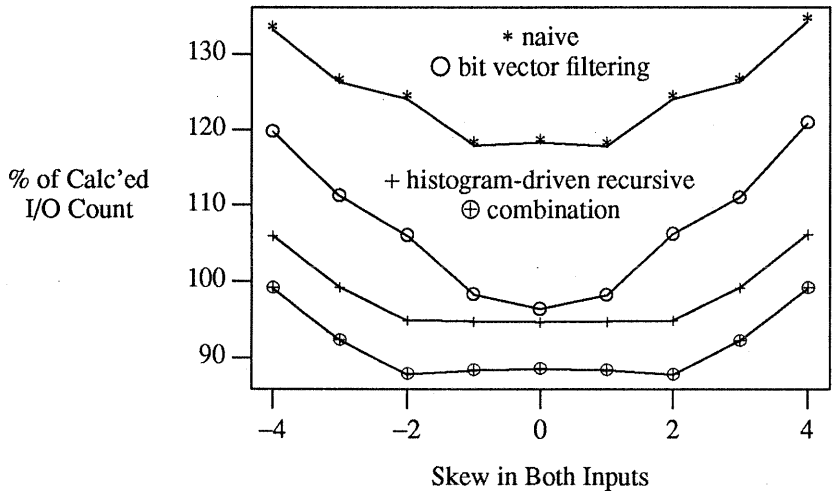


Figure 15. Performance for Similarly Skewed Inputs.

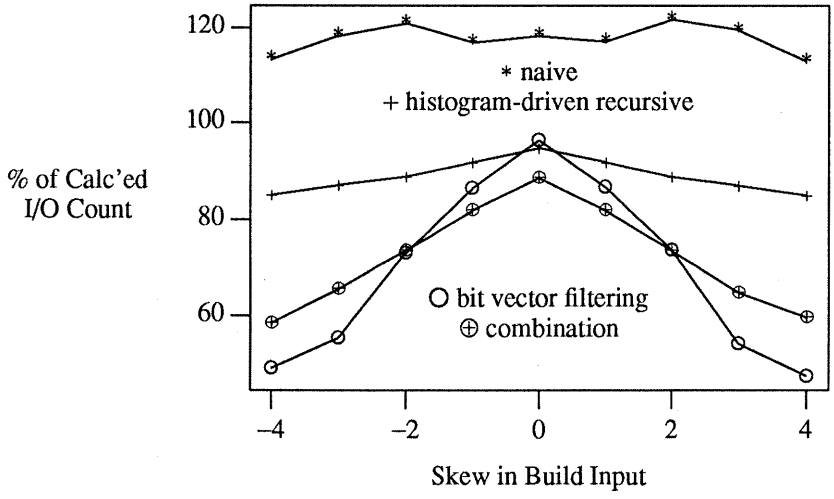


Figure 16. Performance for Differently Skewed Inputs.

skew can also be seen for histogram-driven recursive hybrid hash join, where it creates both opportunities for judicious assignments of buckets to memory and to output partitions and creates the filter effect of histogram-driven recursive hybrid hash join for empty virtual subpartitions. As before, however, the combined algorithms is competitive or superior to either pure hybrid hash join enhancement, with the exception for extreme skews in opposite directions.

Summary of Skew Management

To summarize our findings on skew and on histogram-driven recursive hybrid hash join, skew can result in serious performance degradation if it is not controlled and managed in the hybrid hash join algorithm. Histograms are one means to manage skew, which was shown to be quite effective in histogram-driven recursive hybrid hash join. In fact, if managed carefully, data skew can actually improve the performance of hybrid hash join. While naive hybrid hash join and bit vector filtering are useful for skew in the build input, histogram-driven recursive hybrid hash join manages and exploits both build and probe input skew. The performance of histogram-driven recursive hybrid hash join can be further improved by combining it with bit vector filtering in each recursion level. The presented simulations do not include the option of role reversal for individual virtual subpartitions, which may lead to additional performance gains for histogram-driven recursive hybrid hash join.

7. Multi-Way Joining

Beyond robustness in situations with unpredictable relative input sizes and with data and hash value skew, the third issue traditionally seen as crucial advantage of merge-join over hybrid hash join is the ability to exploit "interesting orderings" for queries with multiple joins on one attribute [33]. A strong argument in favor of sorting and merge-join is the fact that merge-join delivers its output in sorted order; thus, multiple merge-joins on the same attribute can be performed without sorting intermediate results between merge-join operators.

For binary matching operations that consider all attributes, i.e., intersection, union, and difference, any input order is interesting. For example, the intersection of two relations can be formed using an intersection algorithm based on merge-join using any sort order, as long as the two sort orders are the same. Thus, for set operations, the advantage of sorting and merging over hashing does not depend on the query predicate. In other words, while this section pertains to multiple joins only if they use a common join attribute, it pertains to any sequence of set operations.

Algorithm Discussion

For joining three relations, as shown in Figure 17, pipelining data from one merge-join to the next without sorting translates into a 3:4 advantage in the number of sorts compared to two joins on different join keys. For joining N relations on the same key, only N sorts are required instead of $2 \times N - 2$ for joins on different attributes. Of course, the size of the intermediate result O_1 relative to the inputs I_1, I_2, and I_3 can make this savings in elapsed time either larger or smaller than 3:4.

Hash-based algorithms tend to produce their outputs in a very unpredictable order, i.e., depending on the hash function and on overflow management. In order to take advantage of multiple joins on the same attribute, the equality has to be considered in the step of hash-based algorithms that is controlled by key values, i.e., during partitioning. In other words, such join queries could be executed effectively by a hash join algorithm that has N

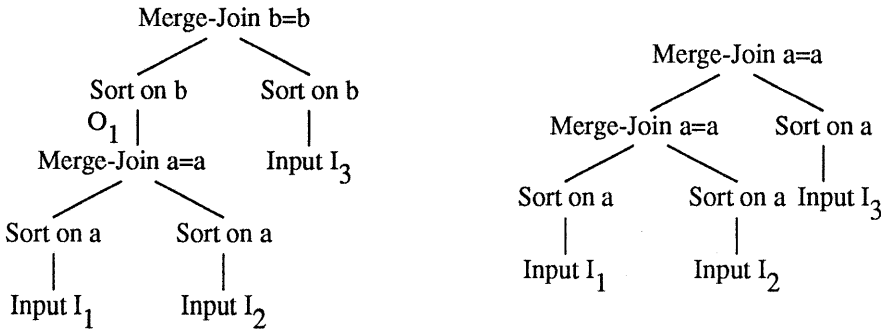


Figure 17. The Effect of Interesting Orderings.

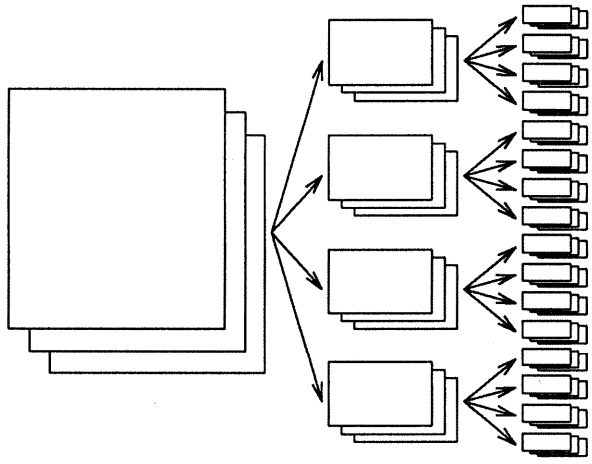


Figure 18. Partitioning in a Multi-Way Hash Join.

inputs, partitions them all concurrently, and then recursively performs N-way joins on each N-tuple of partition files, not pairs as in binary hash join with one build and one probe file for each partition. Figure 18 shows the partitioning levels for a 3-way join. In the deepest recursion level, two in-memory hash tables are used (one each for two inputs) and probed with the third input. Figure 19 shows a plan using conventional, binary hybrid hash joins, and an equivalent N-way join operator in a query plan. The N-way hybrid hash join is unusual, because most query algebra operators have one or two inputs, not a variable number. However, the N-way operator is more efficient, because the intermediate result O_1 does not get partitioned and therefore the I/O required for writing and reading partition files for O_1 is saved.

Performance Observations

Consider the I/O costs of the query evaluation plans in the last three figures, i.e., sorting and merge-join with different and equal join attributes and 3-way hash join on the same join attribute. If we assume that the sorts for the first two inputs perform the final merge concurrently with each other and with the run-generation phase of the intermediate sort, and that memory is divided evenly among the three sort processes in this phase, the I/O cost for input I_1 is approximately

$$2 \times I_1 \times \left\lceil 1 + \log_F \left\lceil I_1 / (F / 3) / M \right\rceil \right\rceil.$$

The final merge level appears as the constant 1 in the formula, the fact that the earlier merge levels create not one but $F / 3$ runs is reflected in the division by $F / 3$. The formula for input I_2 is similar to that for input I_1. The I/O cost for input I_3 is, with the final merge using one half of memory,

$$2 \times I_3 \times \left\lceil 1 + \log_F \left\lceil I_3 / 2 / M \right\rceil \right\rceil.$$

For the intermediate result of size O_1, with run generation using one third and final merge using one half of memory, the I/O cost is approximately

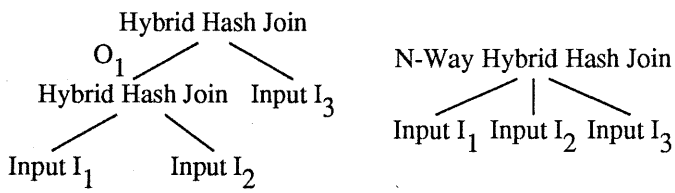


Figure 19. The N-Way Hash Join Operator in a Query Plan.

$$2 \times O_1 \times \left\lceil 1 + \log_F \left\lceil O_1 / 2 / (M / 3) \right\rceil \right\rceil .$$

For the merge-join plan with only three sorts, with all final merges using only one third of memory, the costs for inputs I_1 and I_2 are the same as above. The cost for input I_3 is also calculated with the formula for input I_1 above. Thus, the total I/O cost for the merge-join plan exploiting interesting orderings is

$$\sum_{i=1}^{3} 2 \times I_i \times \left\lceil 1 + \log_F \left\lceil I_i / (F / 3) / M \right\rceil \right\rceil ,$$

which can be simplied to

$$= 2 \times \sum_{i=1}^{3} I_i \times \log_F \left\lceil 3 \times I_i / M \right\rceil$$

and generalized for arbitrary N to

$$= 2 \times \sum_{i=1}^{N} I_i \times \log_F \left\lceil N \times I_i / M \right\rceil .$$

For the hash-based N-way join shown in Figure 19, recursion terminates when two of the three partition files fit into memory. Presuming inputs I_1 and I_2 are smaller than input I_3, $\log_F ((I_1 + I_2) / M)$ levels of recursion are required. Thus, the total I/O cost for this 3-way join operation is

$$2 \times \sum_{i=1}^{3} I_i \times \log_F \left\lceil (I_1 + I_2) / M \right\rceil .$$

Generalized for arbitrary N and slightly transformed, this is

$$= 2 \times \sum_{i=1}^{N} I_i \times \log_F \left\lceil \sum_{i=1}^{N-1} I_i / M \right\rceil ,$$

and if indeed the largest input is chosen as probe input, the cost of N-way hash-based matching is

$$= 2 \times \sum_{i=1}^{N} I_i \times \log_F \left\lceil \left(\sum_{i=1}^{N} I_i - \max_{i=1,..,N} I_i \right) / M \right\rceil .$$

This cost is preferable over the cost of the sort-based plan with direct pipelining between two merge-joins for two reasons. First, the recursion depth is not influenced at all by the size of one of the inputs. Ensuring that this be the largest input, either within the query optimizer or by dynamic role reversal discussed in the previous section, makes hash-based N-way matching of inputs of different sizes more efficient than sort-based matching.

Second, in the deepest recursion level, items from only two, not three (or $N - 1$ vs. N in the general case) inputs must be in memory. Thus, the number of recursion levels in hash-based matching can be expected to be slightly less than the number of merge levels in sort-based matching [18]. Thus, for matching N inputs of equal size, hash-based N-way matching will outperform sort-based strategies. However, as also pointed out in [18], using replacement selection instead of quicksort for run generation results in larger and fewer runs (by a factor of 2) and therefore a slightly reduced number merge levels (by $1 / \log_2 F$ [16]).

Where it applies, N-way matching eliminates partitioning of intermediate results. Matching N inputs using binary algorithms uses $N - 1$ intermediate results. Thus, if all inputs and intermediate results are of the same size, the N-way matching algorithm improves hash-based matching by a factor of $(N + N - 1) / N = 2 - 1 / N$. Thus, under this assumption, a query evaluation plan based on sorting, merge-join, and exploiting interesting orderings will not be faster than an equivalent N-way hybrid hash join operator. However, if one of the N inputs is particularly large, hash-based N-way matching will outperform sort-based matching because one of the inputs does not influence the recursion depth, which can be the largest of the N inputs if role reversal is used.

While general role reversal in N-way matching is very cumbersome to implement, the cost formulas suggest that it is not required for combining the advantages of role reversal and N-way matching. Since only the last input plays a special role in N-way matching, namely as probe input for $N - 1$ hash tables, dynamic role assignment only requires that any input can be used for probing. Since the order in which the $N - 1$ hash tables are probed is not significant for the performance of N-way matching, the implementation techniques required for $N - 1$ binary operations with role reversal suffice to realize sufficiently general role reversal facilities in hash-based N-way matching.

Summary and Final Remarks on Multi-Way Joins

In summary, interesting orderings are very useful in relational join as well as for binary set operations such as intersection, and have been considered a major argument for using sorting and merge-join as the main algorithms in many database query processing systems. However, by considering not only binary but also N-way recursive hybrid hash join, we have replicated the advantages of interesting orderings for merge-join for hash-based query processing. In fact, hash-based N-way hybrid hash join can be faster than merge-join, because the largest of the N inputs neither influences the recursion depth nor requires memory for a hash table.

As a final remark on interesting orderings, B-tree indices and merge-join are sometimes preferred over hash indices and hybrid hash join because B-tree index scans can deliver data in an interesting ordering, and performance handbooks for relational database system products suggest creating and maintaining B-tree indices on keys and foreign keys to speed merge-join and index nested loops join. This argument, however, is circular, if B-tree indices justify merge-join and merge-join justifies B-tree indices. While it is a good idea to index join attributes, the decision which type of index structure to use should be based on whether the underlying domain is ordered or not. For ordered domains for which users can specify $\leq$ predicates, B-tree indices are most useful. However, for unordered domains like most artificial keys, e.g., social security numbers, hash indices should be used. Considering that most joins use a key and a foreign key, i.e., both join attributes are from the same domain, pairs of hash indices permit efficient joins by "merging" the index leaves by hash values. The order in which join results are produced is a "hash-ordering" somewhat similar to a merge-join on compressed values.

8. Summary and Conclusions

In this paper, we have explored performance enhancements for hybrid hash join. Join was used as a representative of all binary matching operations including intersection, union, difference, semi-join, and outer join. Since these operations must be supported in any data model and database management system used for large data volumes, our analysis and performance improvements pertain not only to relational but also to extensible and object-oriented database systems.

We considered five specific techniques for improving hybrid hash join performance. First, data compression by domain permits superlinear speedups, mostly without requiring decompressing intermediate results. Second, large clusters make partitioning significantly faster. These first two techniques are usable both for sort- and hash-based query processing algorithms and can each contribute a factor of 2 to 5 to query processing performance. Third, reversing the roles of build and probe inputs makes hybrid hash join performance independent of whether or not the query optimizer can reliably predict the inputs' relative sizes. In extreme cases, role reversal improves query processing performance by a factor of 5 or more. Fourth, using histogram-driven recursive hybrid hash join eliminates and even reverses undesirable effects of hash value skew. For inputs with duplicate- or distribution-skew, histogram-driven recursive hybrid hash join can be up to 2 times faster than the standard hybrid hash join, sometimes even more. Fifth, partitioning multiple inputs simultaneously reduces the number of intermediate results that must be partitioned, and may contribute another factor of 2 to query processing performance. Together, the five techniques presented in this paper make hash-based query processing significantly faster. Moreover, all the performance enhancements discussed in this paper can be freely combined with parallelism on shared-, distributed-, and hierarchical-memory architectures.

Earlier research [18], recommended sorting and merge-join over hybrid hash join in the cases of (i) the presence or danger of hash value skew (including skew created by duplicate data values), (ii) the query optimizers' inability to determine the inputs' relative sizes a priori in complex queries, or (iii) complex query predicates using the same join attribute, i.e., queries that permit exploiting "interesting orderings" [33] and pipelining intermediate results from one merge-join to the next without intermediate sort. The case of hash value skew is not only mitigated but actually exploited by the new histogram-driven recursive hybrid hash join algorithm. Queries that can exploit interesting sort orderings can be executed using the new N-way hash-based matching that is even more efficient than multiple pipelined merge-join operators. Unpredictable relative input sizes can be managed

effectively by reversing the roles of build and probe inputs in binary operations and by changing the ordering in N-way matching. Thus, the main conclusion of this research is that if these new techniques are applied in a query execution system, histogram-driven recursive hybrid hash join and its variants for the other matching operations will always dominate sort-based matching.

The present paper has left a number of questions unanswered, for example:

(1) At which processor-to-disk speed ratio does compression become a viable overflow resolution method? Does it depend on input and memory sizes?

(2) How do cluster size and fan-out considerations change if multiple disks are used for partition files?

(3) Can the cluster-size and fan-out considerations be generalized to latency and bandwidth optimizations [16]? Can these optimizations be augmented with real-world costs in order to obtain a general formula for hardware configurations for database query processing similar to the 5-minute rule for memory and buffer sizes [19]?

(4) How can disk arrays be used most efficiently for partitioning and, by duality [18], for merging? Is treating a set of disk spindles as a single device the most suitable abstraction for partitioning and merging or should each spindle be used as a separate, sequential device? How large is the performance gain of not using parity and redundancy for temporary (i.e., run and partition) files in disk arrays?

(5) What catalog information is required and practical to permit histogram-driven recursive hybrid hash join even in the first partitioning step? Can suitable histograms be obtained by sampling?

(6) What are the precise effects of input sizes, domain size, number of distinct actual values, quality of the hash function, memory size, and fan-out on the performance of histogram-driven recursive hybrid hash join?

(7) How can histogram-driven recursive hybrid hash join be used in unary operations and N-way hybrid hash join, and how effective is it?

(8) How can the presented performance enhancements for hybrid hash join and all other matching operations be exploited in complex query plans with many operators? What are the performance effects of depth-first and breadth-first partitioning [16] in complex query plans?

We are currently working on answers to these questions.

Acknowledgements

David DeWitt and Leonard D. Shapiro made a number of insightful comments on earlier drafts of this paper that have strengthened both the argument and the presentation. — This manuscript is based on research partially supported by the National Science Foundation with grants IRI-8996270, IRI-8912618, and IRI-9116547, DARPA with contract DAAB 07-91-C-Q518, Texas Instruments, Digital Equipment Corp., Intel Supercomputer Systems Division, Sequent Computer Systems, ADP, and the Oregon Advanced Computing Institute (OACIS).

References

[1] T. Bell, I. H. Witten and J. G. Cleary, "Modelling for Text Compression", *ACM Computing Surveys 21*, 4 (December 1989), 557.

[2] M. Blasgen and K. Eswaran, "On the Evaluation of Queries in a Relational Database System", *IBM Research Report RJ 1745*, San Jose, CA, April 8, 1976.

[3] M. Blasgen and K. Eswaran, "Storage and Access in Relational Databases", *IBM Systems Journal 16*, 4 (1977).

[4] K. Bratbergsengen, "Hashing Methods and Relational Algebra Operations", *Proc. Int'l. Conf. on Very Large Data Bases*, Singapore, August 1984, 323.

[5] H. L. Bremers, "Hash Partitioning Performance Improved By Exploiting Skew and Dealing with Duplicates", *M.S. Thesis, University of Colorado at Boulder*, 1991.

[6] J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions", *Journal of Computers and System Science 18*, 2 (1979), 143.

[7] S. Christodoulakis, "Implications of Certain Assumptions in Database Performance Evaluation", *ACM Trans. on Database Systems 9*, 2 (June 1984), 163.

[8] D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker and D. Wood, "Implementation Techniques for Main Memory Database Systems", *Proc. ACM SIGMOD Conf.*, Boston, MA, June 1984, 1.

[9] D. J. DeWitt and R. H. Gerber, "Multiprocessor Hash-Based Join Algorithms", *Proc. Int'l. Conf. on Very Large Data Bases*, Stockholm, Sweden, August 1985, 151.

[10] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine", *Proc. Int'l. Conf. on Very Large Data Bases*, Kyoto, Japan, August 1986, 228. Reprinted in M. Stonebraker, Readings in Database Systems, Morgan-Kaufman, San Mateo, CA, 1988.

[11] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. I. Hsiao and R. Rasmussen, "The Gamma Database Machine Project", *IEEE Trans. on Knowledge and Data Eng. 2*, 1 (March 1990), 44.

[12] D. J. DeWitt, J. F. Naughton, D. A. Schneider and S. Seshadri, "Practical Skew Handling in Parallel Joins", *Proc. Int'l. Conf. on Very Large Data Bases*, Vancouver, BC, Canada, 1992.

[13] D. Dewitt, J. Naughton and D. Schneider, "Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting", *Proc. Int'l. Conf. on Parallel and Distributed Information Systems*, Miami Beach, Fl, December 1991.

[14] S. Fushimi, M. Kitsuregawa and H. Tanaka, "An Overview of The System Software of A Parallel Relational Database Machine GRACE", *Proc. Int'l. Conf. on Very Large Data Bases*, Kyoto, Japan, August 1986, 209.

[15] G. Graefe, "Parallel External Sorting in Volcano", *CU Boulder Computer Science Technical Report 459*, 1990.

[16] G. Graefe, "Query Processing Techniques for Large Databases", *submitted for publication*, 1992. Also CU Boulder Computer Science Technical Report 579.

[17] G. Graefe, "Volcano, An Extensible and Parallel Dataflow Query Processing System", *to appear in IEEE Trans. on Knowledge and Data Eng.*, 1993. A more detailed version is available as CU Boulder Computer Science Technical Report 481, July 1990.

[18] G. Graefe, A. Linville and L. D. Shapiro, "Sort versus Hash Revisited", *to appear in IEEE Trans. on Knowledge and Data Eng.*, 1993. An earlier version is available as CU Boulder Computer Science Technical Report 534, July 1991.

[19] J. Gray and F. Putzolo, "The 5 Minute Rule for Trading Memory for Disc Accesses and The 10 Byte Rule for Trading Memory for CPU Time", *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, 395.

[20] Y. E. Ioannidis and S. Christodoulakis, "On the Propagation of Errors in the Size of Join Results", *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 268.

[21] T. Keller and G. Graefe, "The One-to-One Match Operator of the Volcano Query Processing System", *Oregon Graduate Center, Computer Science Technical Report*, Beaverton, OR, June 1989.

[22] M. Kitsuregawa, H. Tanaka and T. Motooka, "Application of Hash to Data Base Machine and Its Architecture", *New Generation Computing 1*, 1 (1983).

[23] M. Kitsuregawa, M. Nakayama and M. Takagi, "The effect of bucket size tuning in the dynamic hybrid GRACE hash join method", *Proc. Int'l. Conf. on Very Large Data Bases*, Amsterdam, The Netherlands, 1989, 257.

[24] M. Kitsuregawa and Y. Ogawa, "Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Skew in the Super Database Computer (SDC)", *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, 1990, 210.

[25] R. P. Kooi, "The Optimization of Queries in Relational Databases", *Ph.D. Thesis, Case Western Reserve University*, September 1980.

[26] R. P. Kooi and D. Frankforth, "Query Optimization in Ingres", *IEEE Database Eng. 5*, 3 (September 1982), 2.

[27] D. A. Lelewer and D. S. Hirschberg, "Data Compression", *ACM Computing Surveys 19*, 3 (September 1987), 261.

[28] M. Nakayama, M. Kitsuregawa and M. Takagi, "Hash-Partitioned Join Method Using Dynamic Destaging Strategy", *Proc. Int'l. Conf. on Very Large Data Bases*, Long Beach, CA, August 1988, 468.

[29] E. Omiecinski, "Performance Analysis of A Load Balancing Relational Hashing Join Algorithm for a Shared- Memory Multiprocssor", *Proc. Int'l. Conf. on Very Large Data Bases*, Barcelona, Spain, 1991.

[30] D. A. Patterson, G. Gibson and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proc. ACM SIGMOD Conf.*, Chicago, IL, June 1988, 109.

[31] D. Schneider and D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment", *Proc. ACM SIGMOD Conf.*, Portland, OR, May-June 1989, 110.

[32] D. Schneider, "Complex Query Processing in Multiprocessor Database Machines", *Ph.D. Thesis, Computer Sciences Technical Report 965*, 1990.

[33] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, "Access Path Selection in a Relational Database Management System", *Proc. ACM SIGMOD Conf.*, Boston, MA, May-June 1979, 23. Reprinted in M. Stonebraker, Readings in Database Systems, Morgan-Kaufman, San Mateo, CA, 1988.

[34] S. Seshadri and J. F. Naughton, "Sampling Issues in Parallel Database Systems", *Proc. Int'l. Conf. on Extending Database Technology*, Vienna, Austria, March 1992.

[35] L. D. Shapiro, "Join Processing in Database Systems with Large Main Memories", *ACM Trans. on Database Systems 11*, 3 (September 1986), 239.

[36] E. J. Shekita and M. J. Carey, "A Performance Evaluation of Pointer-Based Joins", *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 300.

[37] C. Walton, A. Dale and R. Jenevein, "A Taxonomy and Performance Model of Data Skew in Parallel Joins", *Proc. Int'l. Conf. on Very Large Data Bases*, Barcelona, Spain, 1991.

[38] H. Zeller and J. Gray, "An Adaptive Hash Join Algorithm for Multiuser Environments", *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, 1990, 186.