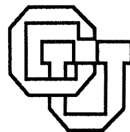
Sort versus Hash Revisited

Goetz Graefe, Ann Linville, Leonard D. Shapiro

CU-CS-534-91 July 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE NATIONAL SCIENCE FOUNDATION

Sort versus Hash Revisited

Goetz Graefe, Ann Linville, Leonard D. Shapiro

CU-CS-534-91 July 1991

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

(303) 492-7514
(303) 492-2844 Fax
graefe@cs.colorado.edu

Sort versus Hash Revisited

Goetz Graefe, Ann Linville, University of Colorado at Boulder
Leonard D. Shapiro, Portland State University

Abstract

Many query processing operations can be implemented using sort- or hash-based algorithms, e.g., join, intersection, and duplicate elimination. In the early relational database systems, only sort-based algorithms were employed. In the last decade, hash-based algorithms have gained acceptance and popularity, and are frequently considered generally superior to sort-based algorithms such as merge join. In this report, we compare sort- and hash-based query processing algorithms using the Volcano query execution engine and conclude that (a) many dualities exist between the two types of algorithms, (b) their costs differ mostly by percentages rather than factors, (c) special cases exist that favor one or the other choice, and (d) there is a strong reason why both sort- and hash-based algorithms should be available in a query processing system.

1. Introduction

With the emergence of relational query languages and algebra, database systems required algorithms to operate on large sets, e.g., for join, intersection, union, aggregation, and duplicate elimination. In early research and implementation efforts, e.g., Ingres [15, 30, 40], System R [2, 6], PRTV [42], and ABE [28], only sort-based methods were employed, and sort costs were one (or even *the*) major component of query processing costs. Consequently, ordering of stored relations and intermediate query processing results were an important consideration in query optimization and led to the concept of *interesting orderings* in System R [38].

While set processing was based on sorting, even early systems employed hash-based algorithms and data structures in form of hash indices [40]. Only in the last decade have hash-based query processing algorithms gained interest, acceptance, and popularity, in particular for relational database machines such as Grace [17, 26] and GAMMA [10, 12], but also for sequential query execution engines [8, 39]. Reasons why hash-based algorithms were not considered earlier include that large main memories are required for optimal performance, and that techniques for avoiding or resolving hash table overflow were needed, i.e., algorithms to handle the case that none of the sets to be processed fits in main memory.

Hash-based algorithms are now widely viewed as significantly faster than their sort-based equivalents, and many major database system vendors are hurrying to incorporate hash join and aggregation into their products, e.g., [43]. Furthermore, hash-based algorithms are frequently associated with parallel query processing and linear speedup, even though hash-based partitioning of data to several processors can also be combined

with sort-based local algorithms as the Teradata machine proves [41]. In fact, the choices of partitioning and local processing methods are independent or orthogonal from one another.

In this report, we compare sort- and hash-based algorithms, and argue, contrary to "current wisdom," that (a) many dualities exist between the two types of algorithms, (b) their costs differ mostly by percentages rather than factors, (c) special cases exist that favor one or the other choice, and (d) there is a strong reason why both sort- and hash-based algorithms should be available in a query processing system.

The remainder of this report is organized as follows. We discuss sort- and hash-based algorithms as used in real systems or proposed in the literature in Section 2. In Section 3, we consider dualities and differences between sort- and hash-based query processing algorithms. An experimental comparative study of sort- and hash-based join algorithms follows in Section 4. Section 5 contains a summary and our conclusions.

2. Related Work

After the investigations of Blasgen and Eswaran [6, 7], merge join was universally regarded as the most efficient join method for large input files. After sorting both join inputs on the join attribute, tuples with matching join attribute values can be found efficiently and without much memory, independently of the file sizes.

Significant effort has been spent on devising and improving sort algorithms for database systems; recent work includes [1, 35]. The main memory algorithms employed in all these studies are either quicksort or replacement selection; the variations and new ideas mainly concern optimizing the I/O cost of writing and merging temporary files or *runs* by considering larger units of I/O than pages at the expense of smaller merge fan-in. Larger units of I/O allow for faster I/O because the number of seek operations and rotational latencies is reduced. However, since one input buffer is required for each input run during merging, the *fan-in* (number of runs merged simultaneously) is decreased with larger units of I/O. Considering that the number of merge levels, i.e., the number of times each record is merged from one run into another, is the logarithm of the number of initial runs using the fan-in as base, the number of merge levels may increase with reduced fan-in. The most interesting recent insight was that it may be beneficial to use larger units of I/O even if the fan-in is decreased and the number of merge levels is increased [19, 35].

Another important optimization for sorting concerns the merge strategy. Let us explain it with an example shown in Figure 1. Consider a sort with a maximal fan-in of 10 and an input file that requires 12 initial runs. Instead of merging only runs of the same level, it is better to delay merging until the end of the input has been reached, then to merge first 3 of the 12 runs, and finally to merge the output with the remaining 9 runs, as shown in Figure 2. The I/O cost (measured by how many memory loads of data must be written to disk to any of the runs created) for the first strategy is 12+10+2 = 24, while for the second strategy it is 12+3 = 15, meaning that the first strategy requires 60% more I/O than the second one. The general rule is to merge just the right number of runs after the end of the input file has been reached, and to always merge the smallest runs

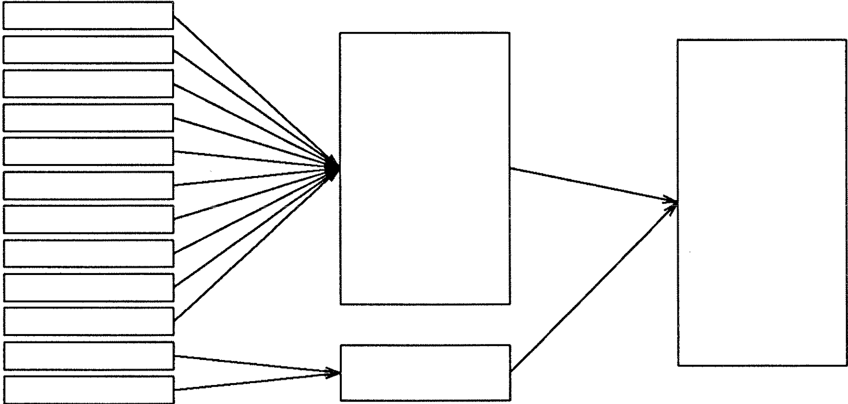


Figure 1. Naive Merging.

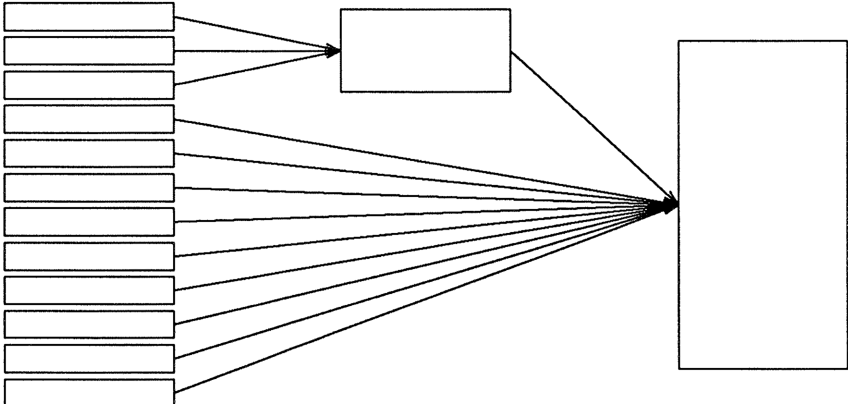


Figure 2. Optimized Merging.

available for merging. More detailed examples are given in [19].

Recently, parallel sorting has found increased interest, e.g. [3, 5, 19, 20, 24, 31, 32, 36]; most investigations concern either clever designs for parallel merging or for partitioning data evenly across a set of machines to achieve good load balancing. In this report, we do not concern ourselves much with parallelism because we believe that the issues of data manipulation and parallelism can be made orthogonal [18, 22] and that our conclusions are directly applicable to algorithms used in parallel environments.

For duplicate elimination and aggregate functions, e.g., sum of salaries by department, Epstein's work has led to the use of sorting for aggregation, too [15]. Aggregation and grouping are frequently assumed to require sorting. It is interesting to note that sorting for aggregation permits a clever optimization [4]. Instead of sorting the input file completely and then combining (adjacent) duplicates, aggregation can be done *early*, namely whenever two records with matching grouping attributes are found while writing a run file. Consider an aggregation with 100,000 input records being aggregated into 1,000 groups using a sort with maximal fan-in 10. If aggregation is not done after sorting, the largest run file may contain 10,000 records. If aggregation is done early, the largest run file will contain at most 1,000 records. In other words, with early aggregation, no run file will ever contain more records than the final output file. If the *reduction factor* (output over input size) is larger than the maximal fan-in, significant improvements can be realized. In the extreme case, if replacement selection is used for creating initial runs and the output (not the input) fits into memory, the entire sort may be accomplished without any run files on disk.

Starting in about 1983, query processing algorithms based on hashing experienced a sudden surge of interest [8, 10, 11, 26], predominantly for relational join. Since they were used in a number of relational database machines, hash-based join algorithms were frequently identified with parallel query execution [12, 17] even though they make equal sense in sequential environments. In its simplest form, called *classic hash join* in [39], a join algorithm consists of two phases. First, an in-memory hash table is built using one input (typically the smaller input) hashed on the join attribute. Second, tuples from the second input are hashed on their join attribute and the hash table is probed for matches.

The various forms of hash join differ mainly in their strategies for dealing with *hash table overflow*, i.e., the case that the smaller input (and therefore the hash table) is larger than main memory. All overflow strategies use overflow files, either one per input or many partition files for each input [11]. Overflow *avoidance*

as used in the Grace database machine [17] builds the overflow files before any overflow actually occurs. Overflow *resolution* creates overflow files after it has occurred. A clever combination of in-memory hash table and overflow resolution called *hybrid hash join* [10, 39] optimizes the I/O for overflow files by retaining as much as possible of the first input relation in memory, i.e., one of the partition files is kept in memory and probed immediately as the other input is partitioned. If the partition or overflow files are still larger than memory, they can be partitioned further using a recursive algorithm until classic or hybrid hash join can be applied.

Hashing can also be used for aggregation and duplicate elimination by finding duplicates while building the hash table. It is interesting to note that overflow occurs only if the output does not fit into main memory, independently of the size of the input. Once overflow occurs, however, input records have to be written to overflow files, including records with duplicate keys that eventually will have to be combined.

3. Duality of Sorting and Hashing

In this section, we outline the similarities and duality of sort- and hash-based algorithms, but also point out where the two types of algorithms differ. We try to discuss the approaches in general terms, ignoring whether the algorithms are used for relational join, union, intersection, aggregation, duplicate elimination, or other operations. Where appropriate, however, we indicate specific operations.

Table 1 gives an overview of the features that correspond to one another. Both approaches allow for in-

Sorting	Hashing
Quicksort	Classic Hash
Physical divide, logical combine	logical divide, physical combine
Single-level merge	Partitioning into overflow files
Sequential write, random read	Random write, sequential read
Fan-in	Fan-out
Multi-level merge	Recursive overflow resolution
Number of merge levels	Recursion depth
Non-optimal final fan-in	Non-optimal hash table size
Merge optimizations	Bucket tuning
Reverse runs & LRU	Hybrid hash
Replacement selection	?
?	Single input in memory
Aggregation in replacement selection	Aggregation in hash table
Interesting orderings	N-way joins, hash-merging

Table 1. Duality of Sort- and Hash-Based Algorithms.

memory versions for small data sets and disk-based versions for larger data sets. If a data set fits into memory, quicksort can be employed for sorting and classic (in-memory) hash can be used as hashing technique. It is interesting to note that both, quicksort and classic hash, are also used in memory to operate on subsets after "cutting" an entire large data set into pieces. The "cutting" process is part of the familiar *divide-and-conquer* paradigm employed for both sorting and hashing. There exists, however, an important difference. In the sort-based algorithms, a large data set is divided into subsets using a physical rule, namely into chunks as large as memory. These chunks are later combined using a logical step, merging. In the hash-based algorithms, the large data set is cut into subsets using a logical rule, by hash values. The resulting partitions are later combined using a physical step, simply concatenating the subsets or result subsets. In other words, a single-level merge in a sort algorithm is a dual to partitioning in hash algorithms. Figure 3 illustrates this duality and the opposite directions.

This duality can also be observed in the behavior of a disk arm performing the I/O operations for merging or partitioning. While writing initial runs after sorting them with quicksort, the I/O is sequential. During merging, read operations access the many files being merged, and require random I/O capabilities. During partitioning, the I/O operations are random, but when reading a partition later on, they are sequential.

For both approaches, sorting and hashing, the amount of available memory limits not only the amount of data in a basic unit processed using quicksort or classic hash, but also the number of basic units that can be

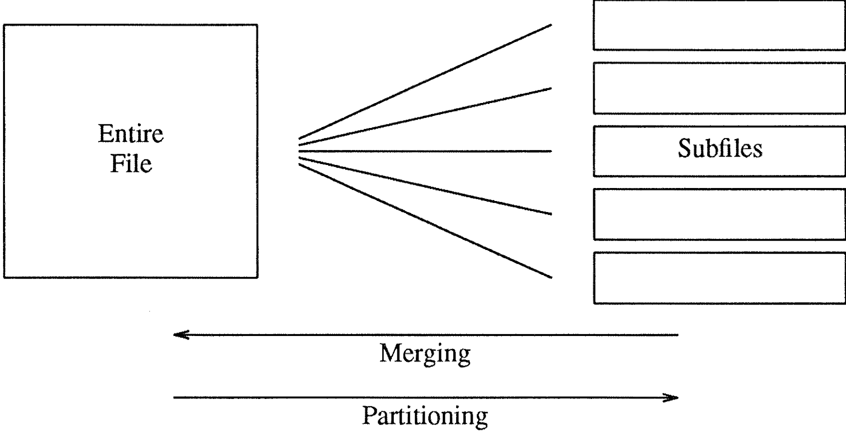


Figure 3. Duality of Partitioning and Merging.

accessed simultaneously. For sorting, it is well known that merging is limited to the quotient of memory size and buffer space required for each run, called the merge *fan-in*. Similarly, partitioning is limited to the same fraction, called the *fan-out*, since the limitation is encountered while writing partition files. Considering this limitation on fan-in and fan-out, additional techniques must be used for very large data sets. Merging can be performed in multiple levels, each combining multiple runs into larger ones. Similarly, partitioning can be repeated recursively, i.e., partition files are re-partitioned, the results re-partitioned, etc., until the partition files fit into main memory. During merging, the runs grow in each level by a factor equal to the fan-in. For each recursion step, the partition files decrease in size by a factor equal to the fan-out. Thus, the number of levels during merging is equal to the recursion depth during partitioning. There are two exceptions to be made regarding hash value distribution and relative sizes of inputs in binary operations such as join; we ignore those for now and will come back to them later.

If merging is done in the most naive way, i.e., merging all runs of a level as soon as their number reaches the fan-in, the last merge on each level might not be optimal. Similarly, if the highest possible fan-out is used in each partitioning step, the partition file in the deepest recursion level might be smaller than memory, and less than the entire memory is used when processing these files. Thus, in both approaches the memory resources are not used optimally in the most naive versions of the algorithms.

In order to make best use of the final merge (which, by definition, includes all output items and is therefore the most expensive merge), it should proceed with the maximal possible fan-in. This can be ensured by merging fewer runs than the maximal fan-in after the end of the input file has been reached (as illustrated in the previous section). There is no direct dual in hash-based algorithms for this optimization. With respect to memory utilization, the fact that a partition file and therefore a hash table might actually be smaller than memory is the closest to a dual. Utilizing memory more effectively and using less than the maximal fan-out in hashing has been addressed in research on bucket tuning [27].

The development of hybrid hash algorithms [10, 39] was a logical consequence of the advent of large main memories that had led to the consideration of hash-based join algorithms in the first place. If the data set is only slightly larger than the available memory, e.g., 10% larger or twice as large, much of the input can remain in memory and is never written to a disk-resident partition file. To obtain the same effect for sort-based algorithms, if the database system's buffer manager is sufficiently smart or receives and accepts

appropriate hints, it is possible to retain some or all of the pages of the last run written in memory and thus achieve the same effect of saving I/O operations. This can be done particularly easily if the initial runs are written in reverse (descending) order and scanned backward for merging. However, if one does not believe in buffer hints or prefers to absolutely ensure desired I/O savings, using a final memory-resident run explicitly in the sort algorithm and merging it with the disk-resident runs can guarantee this effect.

A well-known technique to improve sort performance is to generate runs twice as large as main memory using a priority heap for replacement selection [29]. If the runs' sizes are doubled, their number is cut in half. Therefore, merging can be reduced to some amount, namely $\log_F(2) = 1 / \log_2(F)$ merge levels. However, if two sort operations feed into a merge join and both final merges are interleaved with the join, each final merge can employ only half the memory, and cutting the number of runs in half (on each merge level, including the last one) allows performing the two final merges in parallel without increasing the merge depth.

The effect of cutting the number of runs in half offsets a disadvantage of sorting in comparison to hashing when used to join (intersect, union) two data sets. In hash-based algorithms, only one of the two inputs resides in and consumes memory beyond a single input buffer, not both as in two final merges concurrent with a merge join.

Heap-based run generation has a second advantage over quicksort; this advantage has a direct dual in hashing. If a hash table is used to compute an aggregate function using grouping, e.g., sum of salaries by department, hash table overflow occurs only if the operation's *output* does not fit in memory. Consider, for example, the sum of salaries by department for 100,000 employees in 1000 departments. If the 1000 result records fit in memory, classic hashing (without overflow) is sufficient. On the other hand, if sorting based on quicksort is used to compute this aggregate function, the input must fit into memory to avoid temporary files[1]. If replacement selection is used for run generation, however, the same behavior as with classic hash is easy to achieve.

The final entry in Table 1 concerns *interesting orderings* used in the System R query optimizer [38] and presumably other query optimizers as well. A strong argument in favor of sorting and merge-join is the fact

[1] A scheme using quicksort and avoiding temporary I/O in this case could probably be devised but would be extremely cumbersome; we do not know of any report or system with such a scheme.

that merge-join delivers its output in sorted order; thus, multiple merge joins on the same attribute can be performed without sort operators between merge-join operators. For joining three relations, this 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 2N–2 for joins on different attributes.

Hash-based algorithms tend to produce their outputs in a very unpredictable order (depending on hash function and on overflow management). To take advantage of multiple joins on the same attribute, the equality has to be considered in the logical step of hashing, i.e., during partitioning on the input side. In other words, such join queries could be executed effectively by a hash join algorithm that has N inputs, partitions them all concurrently, and then 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). However, since such an algorithm is probably cumbersome to implement, in particular if some of the "join" operations are actually semi-join, outer join, set intersection, union, or difference, it might well be that this distinction, joins on the same or on different attributes, determines the right choice between sort- and hash-based algorithms for complex queries.

Another use of interesting orderings is the interaction of (sorted, B-tree) index scans and merge join. While it hasn't been reported explicitly in the literature, it is perfectly possible and reasonable to implement a join algorithm that uses two hash indices (provided the same hash function was used to create the indices) like merge join uses two B-trees. For example, it is easy to imagine "merging" the leaves (data pages) of two extendible hash indices [16], even if the key cardinalities and distributions are very different.

In summary, there exist many dualities between sorting using multi-level merging and recursive hash table overflow management. Since there are so many similarities, it is interesting to compare their costs in detail. This is done in the next section.

4. Experimental Comparison of Sorting and Hashing

In this section, we report on a number of join experiments to demonstrate that the duality of sorting and hashing leads to similar performance in many cases, to illustrate the transfer of optimization ideas from one type of algorithm to the other, and to identify the main decision criteria for the choice between sort-based and hash-based query processing algorithms. A similar study of sort- and hash-based algorithms for aggregation, duplicate elimination, and division algorithms could easily be done but has been omitted here for space

reasons. We first describe the experimental environment and then report on a series of experiments.

4.1. Experimental Environment

The testbed for our experiments was the Volcano extensible and parallel query processing engine [22]. Volcano includes its own file system which is similar to WiSS [9]. Much of Volcano's file system is rather conventional. It provides data files, B^+-tree indices, and bidirectional scans with optional predicates. The unit of I/O and buffering, called a *cluster* in Volcano, is set for each file individually when it is created. Files with different cluster sizes can reside on the same device and can be buffered in the same buffer pool. Volcano uses its own buffer manager and bypasses operating system buffering by using raw devices.

Queries are expressed as complex algebra expressions; the operators of this algebra are query processing algorithms. All algebra operators are implemented as *iterators*, i.e., they support a simple *open-next-close* protocol similar to conventional file scans. Associated with each operator is a *state record*. The arguments for the algorithms, e.g., hash table size or a hash function, are part of the state record. All functions on data records, e.g., comparisons and hashing, are compiled prior to execution and passed to the processing algorithms by means of pointers to the function entry points. There is also an argument passed to each function so that the function can be a generic predicate interpreter with the interpretable code as argument.

Since almost all queries require more than one operator, state records can be linked together by means of *input* pointers. All state information for an iterator is kept in its state record; thus, an algorithm may be used multiple times in a query by including more than one state record in the query. The input pointers are also kept in the state records. They are pointers to a *QEP* structure which includes four pointers to the entry points of the three procedures implementing the operator (*open*, *next*, and *close*) and a state record. An operator does not need to know what kind of operator produces its input, and whether its input comes from a complex query tree or from a simple file scan. We call this concept *anonymous inputs* or *streams*. Streams are a simple but powerful abstraction that allows combining any number of operators to evaluate a complex query. Together with the iterator control paradigm, streams represent the most efficient execution model in terms of time and space for single process query evaluation.

Calling *open* for the top-most operator results in instantiations for the associated state record, e.g., allocation of a hash table, and in *open* calls for all inputs. In this way, all iterators in a query are initiated

Iterator	*Open*	*Next*	*Close*
Print	*open* input	call *next* on input; format the item on screen	*close* input
Scan	open file	read next item	close file
Select	*open* input	call *next* on input until an item qualifies	*close* input
Hash join (without overflow resolution)	allocate hash directory; *open* build input; build hash table calling *next* on build input; *close* build input; *open* probe input	call *next* on probe input until a match is found	*close* probe input; deallocate hash directory
Merge-join	*open* both inputs	get *next* item from input with smaller key until a match is found	*close* both inputs
Sort	*open* input; build all initial run files calling *next* on input and quicksort or replacement selection; *close* input; merge run files until their number is reduced to the fan-in; open the remaining run files	determine next output item; read new item from the right run file	destroy remaining run files

Table 2. Examples of Iterator Functions.

recursively. In order to process the query, *next* for the top-most operator is called repeatedly until it fails with an *end-of-stream* indicator. Finally, the *close* call recursively "shuts down" all iterators in the query. This model of query execution matches very closely the iterator concept in the E programming language design [34] and the algebraic query evaluation system of the Starburst extensible relational database system [23]. Table 2 gives a set of simplified examples for *open*, *next*, and *close* functions of some operators.

Figure 4 shows a simple query plan which might illustrate the interaction of operators and their procedures. Calling *open* on the print operator results in an *open* call on the hash join operator. To load the hash table, hash join *open*s the left file scan, requests all records from the file scan by calling its *next* function, and *close*s it. After calling *open* on the right file scan, the *open* procedures of hash join and then print return. Now the query evaluation plan is ready to produce data. Calling *next* on the print operator results in a *next* call of the hash join operator. To produce an output item, the hash join operator calls *next* on the right input until a match is found that can be returned to the print operator. After formatting the record on the screen, the print operator's *next* functions returns. The query execution driver must call the top-most operator's *next* function repeatedly until it receives an error status. When, in a subsequent *next* call, the right file scan returns an end-

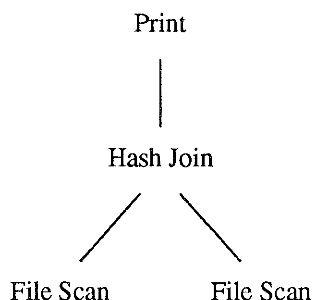


Figure 4. A Simple Query Plan.

of-stream status, the hash join and then the print operators return this status. Query execution completes with a *close* call to the print operator which results in *close* calls for the hash join and the right file scan operators.

Volcano's one-to-one match operator implements all functions in which a record is included in the output depending on the match with another record, namely join, semi-join, outer join, union, intersection, difference, aggregation, and duplicate elimination [25]. It is implemented both sort-based and hash-based. The sort-based version combines a sort operator that includes aggregation and duplicate elimination [19] with a generalized merge-join operator. The hash-based version is a recursive implementation of hybrid hash augmented with aggregation during the build phase and parameterized to allow both overflow avoidance similar to Grace hash join [17] and overflow resolution using the original hybrid hash join [10, 39]. We are currently studying how to incorporate bucket tuning and management of skew into the recursive overflow resolution algorithm.

For creating initial runs in Volcano's sort operator, we decided to use quicksort, not replacement selection, although this technique can create runs larger than memory. The basic idea of replacement selection is that after a record has been written to a run file, it is immediately replaced in memory by another record from the input. Since the new input record can frequently be included in the current output run, runs tend to be about twice as large as memory.

In a page-based environment, however, the advantage of larger initial runs is not without cost. Either the size of the heap used in replacement selection is reduced by about one half to account for record placement and selective retention in input pages (which would offset the expected increase in run length), or a record

holding area and another copying step are introduced. We considered this prohibitively expensive[2], unless the previous query operator must perform a copy step anyway that can be moved into the sort operator, and abandoned the idea of using heaps for creating initial runs. Furthermore, this technique does not work easily for variable-length records.

Volcano is operational on a variety of UNIX machines, including several parallel systems [18, 21]. The experiments were run on a Sun SparcStation running SunOS with two CDC Wren VI disk drives. One disk was used for normal UNIX file systems for system files, source code, executables, etc., while the other was accessed directly by Volcano as a raw device.

4.2. Joins with Equal Input Sizes

In order to demonstrate the relative performance of sorting and merge-join vs. hybrid hash join, we repeatedly joined two relations similar to the Wisconsin benchmark [14]. The two relations had the same cardinality, and each tuple was 208 bytes long. The join attribute was a four-byte integer; each value between 1 and the relation cardinality appeared exactly once. Each tuple (in either relation) had exactly one match in the other relation, and the join result cardinality was equal to each input cardinality. The join result tuples were immediately discarded after the join because we were interested in the relative join performance, not in the performance of writing results to disk. The memory allocated for quicksort, for merging, for partitioning, and the hash table was ½ MB. The cluster size (unit of I/O) was 4 KB; therefore, the maximal fan-in or fan-out was 127 (½ MB / 4 KB – 1) and the final fan-out in two sorts feeding into a merge join was 64.

Figure 5 shows the performance for merge-join and hybrid hash join for input sizes between about 2 MB (10,000 tuples) and about 100 MB (500,000 tuples). Sort and merge-join performance is indicated with circles (O), hybrid hash with squares (□). Note that both axes are logarithmic. Table 3 shows the exact values measured and used in Figure 5. The performance is not exactly linear with the input sizes because both algorithms, merge-join and hybrid hash join, require multiple levels of merging or overflow resolution for the larger inputs. For example, for two 30 MB input files, both algorithms require that each record go to disk once. For 100 MB, both algorithms require that a substantial fraction go to disk twice, either in merging or in overflow resolution.

[2] Note that in many recent computer systems have been designed and optimized for a high MIPS number, sometimes without similar performance advances in mundane tasks such as copying [33]. In a shared-memory parallel machine in which bus bandwidth may be scarce, avoiding copying is even more important.

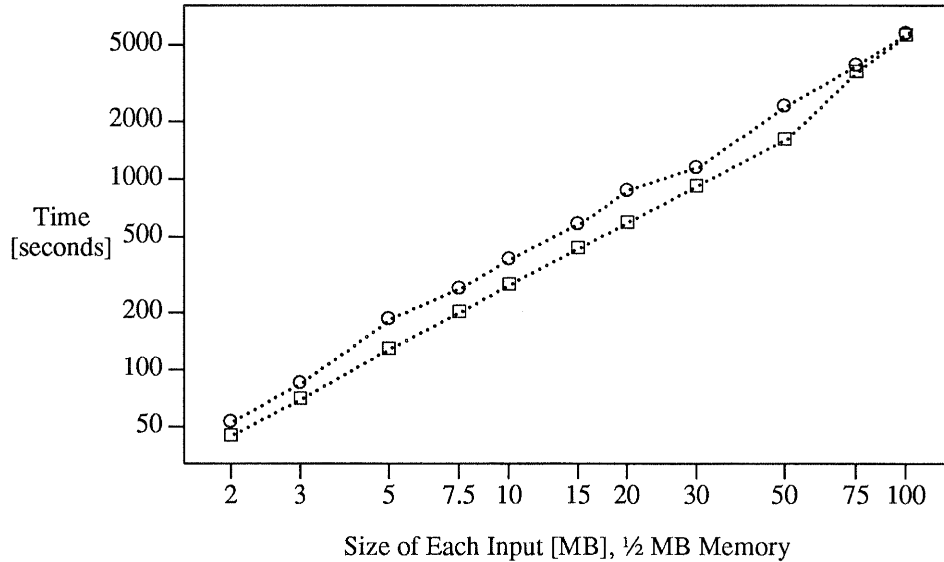


Figure 5. Join Performance for Equal Input Sizes.

Input Size [MB]	Sort-Merge	Hybrid Hash
	[seconds]	
2	53	45
3	85	70
5	184	129
7.5	268	201
10	380	279
15	583	434
20	868	591
30	1145	918
50	2397	1607
75	3917	3631
100	5742	5638

Table 3. Join Performance for Equal Input Sizes.

For 50 MB input files, the advantage of hash-join over merge join is most pronounced because hash join can join the two files with one overflow resolution level (50 MB / 127 ≤ ½ MB), while sorting (using quicksort) with a final fan-in of 64 (two concurrent final merges) requires that some records go to disk twice. This difference between sort- and hash-based join algorithms explains why hash join is about 1.5 times faster than merge join for two input files of this size, but Figure 5 demonstrates that there is only a small range of file sizes for which this difference results in a large performance advantage.

Sorting and merge-join performed worse than hashing in Volcano because Volcano uses quicksort, not replacement selection, for creating initial sorted runs. To illustrate this claim, we calculate the relative I/O

required for sorting using quicksort, sorting using replacement selection, and hybrid hash to join two 50 MB inputs. We only calculate write costs for one input because the I/O is equal for both inputs, and all files written will be read exactly once. Using quicksort, 50 MB of data divided by ½ MB of memory results in 100 runs. As each sort can use a final merge fan-in of 64 (½ MB / 4 KB / 2), 100 runs must be reduced to 64 using a fan-in of 127 (½ MB / 4 KB – 1), requiring 37 (100 – 64 + 1) original runs to be merged into 1 larger run. Thus, the total I/O for sorting with quicksort is proportional to 137 memory loads for each input. For replacement selection, there would have been about 51 runs, each about twice as large as memory for which one final merge would suffice. Thus, the total I/O for sorting with replacement selection is proportional to 100 memory loads for each input. For hybrid hash, the entire inputs have to be partitioned into overflow files of about 0.39 MB (50 MB / 127); hybrid hashing would not be very effective in this situation because at most 27 page buffers (or 108 KB of 50 MB, 0.2%) could have been retained in memory. Each file will fit into memory when joining partition files. Thus, the total I/O will be proportional to the input sizes, or 100 memory loads for each input, exactly the same as for sorting using replacement selection. To summarize these calculations, we chose quicksort in Volcano because it makes the software less complex and performs well with variable-length records even though our measurements would have been better if we had implemented replacement selection.

We would like to discuss why we have obtained different results than Schneider and DeWitt [37] and Shapiro [39]. One reason is that we used a more sophisticated sort operator than was implemented in the GAMMA database machine at the time. GAMMA's sort operator was the same as WiSS' [9], i.e., it sorted from a disk-resident file into a disk-resident file. Therefore, an intermediate result had to be written to disk before it could be sorted rather than sorted into initial runs before the first write step, and the entire sorted file was written back to disk rather than pipelined into the next operation, e.g., a merge-join. Thus, the WiSS sort algorithm can easily require three trips to disk when actually one could have sufficed. Furthermore, neither heap-based run creation nor merge optimizations are implemented in WiSS. Thus, the comparison in [37] is biased against sort-based algorithms. Shapiro [39] analyzed only the case in which hybrid hash's advantage is most pronounced, i.e., when less than one full recursion level is required, based on the argument that most memories are fairly large and multi-level recursion or merging are not realistic. This last argument does not always hold, however, as discussed in the next section.

4.3. Performance Optimizations

In this section, we focus on using duality to transfer tuning ideas from sorting to hashing and vice versa. Originally, the performance of sorting and merge-join in Volcano had been clearly inferior to that of hybrid hash join, in particular for input sizes relatively close to memory size. The big advantage of hybrid hash over overflow avoidance (write all partitions to disk, do not retain any data in memory) is that as much data as possible is never written to temporary files. This led us to search for a dual in the realm of sorting for the in-memory hash table used in hybrid hash join. To obtain the same effect, we changed Volcano's sort operator such that it retains data in memory from the last quicksort for the first merge. In order to achieve that, it writes runs in reverse order, i.e., in descending order for an ascending sort, and gives the *KEEP* hint to the buffer manager for the clusters written after the end of the input has been found. These clusters will stay in the I/O buffer until the first merge, which is ascending and uses a backward scan on the run files. Therefore, these clusters are never written to disk, and a similar effect to hybrid hash join could be achieved. This optimization has been analyzed in some studies, e.g. [39], but was not considered a dual of hybrid hash. Without the focus on duality, we probably would have overlooked it. This optimization makes the most difference for inputs only slightly larger than main memory, precisely the same case when hybrid hash join shows the largest difference to overflow avoidance.

In a recent study of sequential and parallel sorting, we found that the unit of I/O can have a significant impact on sort performance [19] beyond the effect of read-ahead and double buffering [35]. In Volcano, the cluster size is defined for each file individually. Small clusters allow high fan-ins and therefore few merge levels; large clusters restrict the fan-in and may force more merge levels but allow more efficient I/O because more data is moved with each I/O and each merge level can be completed with fewer seeks. For sorting, we found that the optimal performance is typically obtained with a very moderate fan-in and relatively large clusters. If merging and partitioning are indeed duals, we expect the same effect of cluster size on hybrid hash performance.

Figure 6 and Table 4 show the performance of joins of two 20 MB inputs for various cluster sizes. As can be seen, hash performance is as sensitive to cluster size as sorting. The optimal cluster size for hashing seems to be somewhat higher than for sorting; we suspect that this also stems from the fact that merge join and sorting using quicksort uses more merge levels than hash join's recursion levels (the difference is, as noted

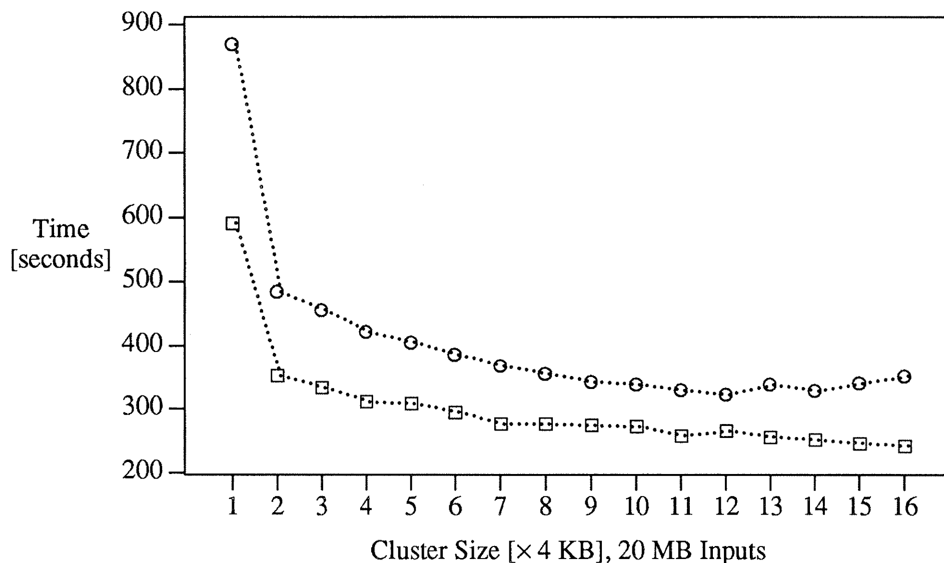


Figure 6. Join Performance by Cluster Size.

Cluster Size [× 4 KB]	Fan-in Fan-Out	Sort-Merge [seconds]	Hybrid Hash [seconds]
1	127	868	590
2	63	483	351
3	41	455	332
4	31	420	310
5	24	404	307
6	20	385	294
7	17	367	275
8	15	354	275
9	13	341	273
10	11	337	271
11	10	328	257
12	9	321	264
13	8	337	255
14	8	327	250
15	7	339	245
16	7	350	241

Table 4. Join Performance by Cluster Size.

earlier, $1 / \log_2 F$). A similar effect was observed in the GAMMA database machine [13], but only for cluster sizes that did not change the recursion depth in hash table overflow resolution. Both algorithms perform best with large cluster sizes and moderate fan-in respectively fan-out, even if multiple merge or recursion levels are required. Around the optimal cluster size, the effect of small changes in the cluster size is fairly small, making a roughly optimal choice sufficient. In an earlier study, we found that the optimal cluster size for sorting depends only on the memory size and not on the input sizes [19]; we suspect the same is true for hashing. In

the following experiments, we used clusters of 32 KB and fan-ins and fan-outs of 15. This is a second optimization we found based on our duality considerations, in this case transfering a sort optimization to hashing.

4.4. Joins with Different Input Sizes

As suggested by Bratbergsengen [8], we decided to include joins of relations with different sizes in the comparison of sorting and hashing. We adjusted the data generation function such that each tuple in the smaller relation has exactly one match in the larger relation. For sorting the inputs of a merge-join, each input determines the number of merge levels for its sort. The large input is merged over more levels than the small input. The only possible optimization we found is the division of memory between the two final merges (of the two inputs) which are overlapped with the actual merge-join. To determine the optimal memory division between two final merges, we approximated the sum of two sort costs with a continuous function and found that the memory allocated to each final merge should be proportional to the size of the inputs. In the following experiments, we divided memory proportionally to the input sizes. For equal input sizes, the two final merge fan-ins were equal; for extremely different sizes, the smaller input is merged into one run such that the final merge is actually just a file scan.

For hashing, the build input determines the recursion depth because partitioning can be terminated as soon as the build partition fits into memory. The recursion depth does not depend at all on the size of the probe input. This is the reason why the smaller of two relations should be chosen to be the build input into a binary hash operation. Changing the role of build and probe input dynamically, e.g., after a first partitioning step, is possible but not considered further in this report.

Figure 7 and Table 5 show the performance of merge-join and hybrid hash join for inputs of equal to very different size. The smaller (build) input size is fixed at 2 MB, the larger (probe) input size is varied between 2 MB and 100 MB[3]. Notice again that both axes are logarithmic. As can be seen, the performance advantage of hybrid hash join increases with the size difference between the input relations. If the probe input is the same size as the build input (2 MB), hash join is faster only because Volcano uses quicksort, as dis-

[3] Notice that the elapsed times for two inputs of 2 MB each are much smaller than in the first experiment, both for sorting and hashing, because we used larger I/O clusters for temporary files (32 KB instead of 4 KB) as suggested in the last subsection.

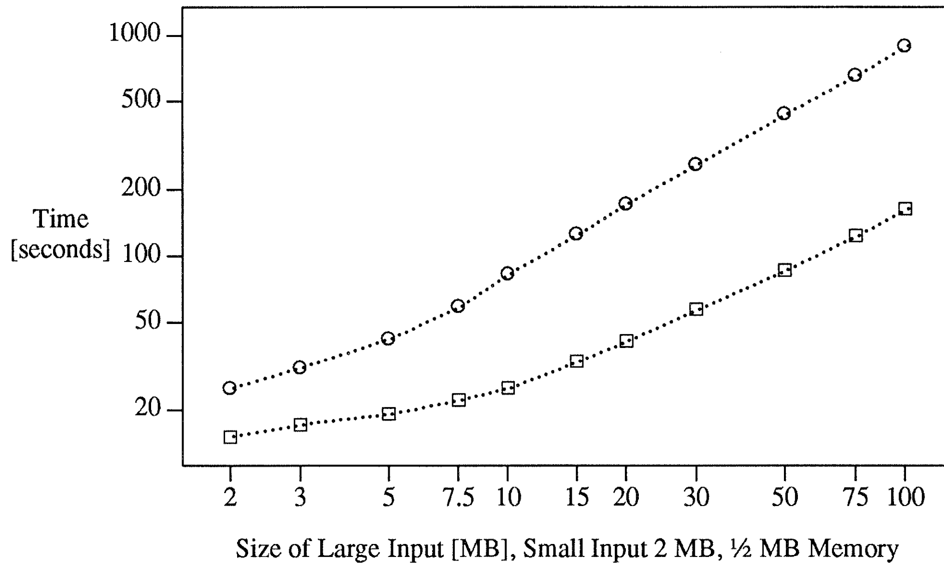


Figure 7. Join Performance for Different Input Sizes.

Large Input [MB]	Sort-Merge [seconds]	Hybrid Hash [seconds]
2	25	15
3	31	17
5	42	19
7.5	59	22
10	83	25
15	125	33
20	172	41
30	260	57
50	439	86
75	655	123
100	887	162

Table 5. Join Performance for Different Input Sizes.

cussed above. For the largest probe inputs, the difference grows to a factor of almost 5.5. The reason is that for hybrid hash join, ¼ of the build relation fits into memory and ¾ of both relations is written to overflow files, independently of the probe input size. For merge-join, sorting the larger input dominates the total cost and makes merge-join the inferior join method for unsorted inputs of very different size.

Similarly, algorithms derived from merge-join for semi-join, outer join, intersection, union, and difference will perform less efficiently than those derived from hybrid hash join for pairs of inputs of very different size. On the other hand, if the query optimizer cannot reliably predict which input is smaller, merge join may be the more robust and therefore superior choice.

4.5. Joins with Skewed Data and Hash Value Distributions

Finally, we experimented with some skewed join value distributions. Instead of using a uniform random number generator to create test data, we used a generalized random function borrowed from Knuth [29]. Using a continuous parameter z, probabilities are assigned to the numbers 1 to N as $P_i = c \;/\; i^z$ for $i=1,...,N$ with $c = 1 \;/ \sum_{j=1}^{N} 1 \;/\; j^z$. If z is 0, the random function creates uniform data; if z is 1, this function can be used to create random data according to Zipf' law [44]. The reason Zipfian distributions are relevant for our purpose is that they were defined to model real data and their frequencies.

Figure 8 shows the probability of values $N = 1,...,100$ with $z = k/5$ for $k=0,...,5$. Since the domain of N is discreet, it is not entirely right to draw the probability functions with continuous lines; however, we have taken the liberty to indicate which data points belong to the same values of z. Note that the y-axis is logarithmic. $z = 0$ is shown by the horizontal line, a uniform distribution. With increasing z, the distribution becomes increasingly skewed. For $z = 1$, the probability values at $N = 100$ is two orders of magnitude higher than for $N = 1$ following Zipf' law. Probabilities with more skew can be obtained with higher values of z.

We used the same data distribution in both inputs. This increases the number of matches between the inputs, resulting in significantly more data copying to create new records and in more backing up in the inner input of merge join.

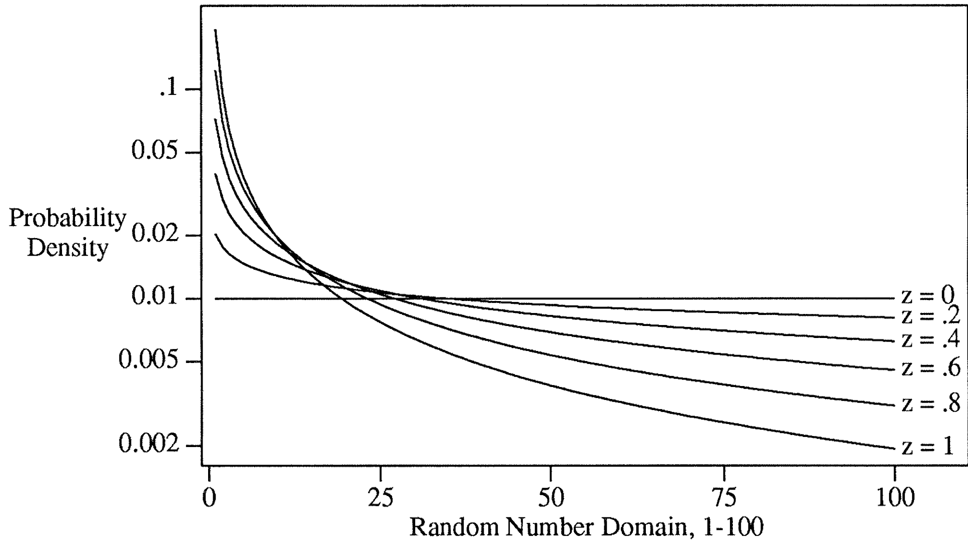


Figure 8. Probability Distributions for Selected Values of z.

Figure 9 and Table 6 show the effect of skew on the performance of merge-join and hybrid hash join. It is evident that merge join is much less affected by the skew. For uniform data, hybrid hash join outperforms merge join, as shown in the previous figures. For highly skewed data, however, merge join outperforms hash join. The reason is that the partitioning is not even; for z equal to 1, ¾ of the entire build (and probe) input is written to a single overflow file (pair). Therefore, instead of performing the join with a single level of overflow resolution, multiple levels are needed.

The reason for this difference between sort- and hash-based algorithms is that sort-based algorithms divide the input file into physical units, i.e., run files are build according to memory size and an input record is

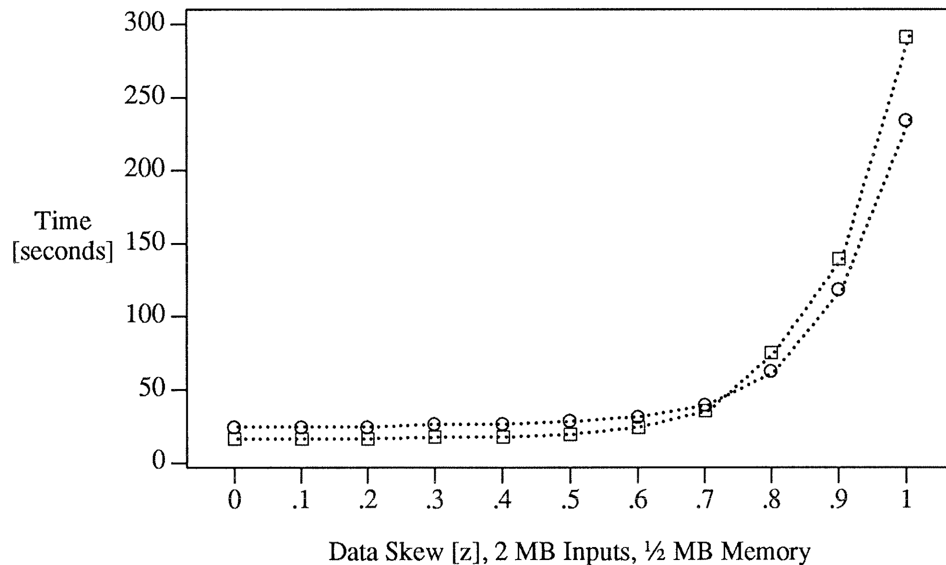


Figure 9. Join Performance for Skewed Data.

Data Skew [z]	Sort-Merge [seconds]	Hybrid Hash
0	24	16
0.1	24	16
0.2	24	16
0.3	26	17
0.4	26	17
0.5	28	19
0.6	31	24
0.7	39	35
0.8	62	75
0.9	118	139
1	234	291

Table 6. Join Performance for Skewed Data.

written to a particular run file solely because of it position in the input, without regard for its sort key. Thus, dividing a sort input into run files is equally efficient for uniform and skewed data. Hashing, however, divides the inputs logically, by hash value. Thus, it is susceptible to skewed hash value distributions. Obviously, skewed hash value distributions are undesirable and against the idea of hashing, i.e., randomizing, the data. To counteract and possibly even exploit hash value skew, we are currently working on a two-step hashing scheme that rerandomizes hash values in hash-based query processing algorithms if a hash function perform unsatisfactorily.

5. Summary and Conclusions

In this report, we have outlined many dualities between sort- and hash-based query processing algorithms, e.g., for join, intersection, or duplicate elimination. Under most circumstances, the cost of optimally implemented algorithms differs by percentages rather than factors. Furthermore, tuning ideas can be transfered from one algorithm group to the other, as we indicated for the cluster size optimizations and concurrent use of temporary files and main memory which is the central idea of hybrid hash algorithms. We expected these results from the large number of dualities and verified them with the Volcano query processing system.

Two special cases exist which favor one or the other, however. First, if two join inputs are of different size (and the query optimizer can reliably predict this difference), hybrid-hash join will outperform merge-join because only the smaller of the two inputs will determine what fraction of the input files will have to be written to temporary disk files during partitioning (or how often each record has to be written to disk during recursive partitioning), while each file determines its own disk I/O in sorting. In other words, sorting the larger of two join inputs is more expensive than writing a small fraction of that file to hash overflow files. This performance advantage of hashing grows with the quotient of the larger over the smaller input file size.

Second, if the hash function is very poor, e.g., because a prior selection on the join attribute or a correlated attribute, hash partitioning can perform very poorly and create significantly higher costs than sorting and merge-join. If the quality of the hash function cannot be predicted or improved (tuned) dynamically, sort-based query processing algorithms are superior because they are less vulnerable to data distributions. Since both cases, join of differently-sized files and skewed hash value distributions, are realistic situations in

database query processing, we recommend that both sort- and hash-based algorithms be included in a query processing engine and chosen by the query optimizer according to the two cases above. If both cases arise simultaneously, i.e., a join of differently-sized inputs with unpredictable hash value distribution, the query optimizer has to estimate which one poses the greater danger to system performance and predictability and choose accordingly.

The important conclusion from this research is that neither the input sizes nor the memory size determine the choice between sort- and hash-based query processing algorithms. Instead, the choice should be governed by the *relative* sizes of the two inputs into binary operators and by the danger of skewed data or hash value distributions. Furthermore, because neither algorithm type outperforms the other in all situations both should be available in a query execution engine for a choice to be made in each case by the query optimizer.

Acknowledgements

The initial interest in comparing sort- and hash-based algorithms in greater detail resulted from a spirited discussion with Bruce Lindsay and Hamid Pirahesh during VLDB 1988. David DeWitt made several very helpful comments. — This work was supported in part by the National Science Foundation with grants IRI-8996270 and IRI-8912618 the Oregon Advanced Computing Institute (OACIS), ADP, Intel Supercomputer Systems Division, and Sequent Computer Systems.

References

1. A. Aggarval and J. S. Vitter, "The Input/Output Complexity of Sorting and Related Problems", *Communications of the ACM 31* (1988), 1116-1127.
2. M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade and V. Watson, "System R: A Relational Approach to Database Management", *ACM Transactions on Database Systems 1*, 2 (June 1976), 97-137.
3. M. Beck, D. Bitton and W. K. Wilkinson, "Sorting Large Files on a Backend Multiprocessor", *IEEE Transactions on Computers 37* (1988), 769-778.
4. D. Bitton and D. J. DeWitt, "Duplicate Record Elimination in Large Data Files", *ACM Transactions on Database Systems 8*, 2 (June 1983), 255-265.
5. D. Bitton Friedland, "Design, Analysis, and Implementation of Parallel External Sorting Algorithms", *Computer Sciences Technical Report 464* (January 1982), University of Wisconsin — Madison.
6. M. Blasgen and K. Eswaran, "On the Evaluation of Queries in a Relational Database System", *IBM Research Report*, San Jose, CA., April 8, 1976.
7. M. Blasgen and K. Eswaran, "Storage and Access in Relational Databases", *IBM Systems Journal 16*, 4 (1977).
8. K. Bratbergsengen, "Hashing Methods and Relational Algebra Operations", *Proceedings of the Conference on Very Large Data Bases*, Singapore, August 1984, 323-333.

9. H. T. Chou, D. J. DeWitt, R. H. Katz and A. C. Klug, ''Design and Implementation of the Wisconsin Storage System'', *Software - Practice and Experience 15*, 10 (October 1985), 943-962.
10. D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker and D. Wood, ''Implementation Techniques for Main Memory Database Systems'', *Proceedings of the ACM SIGMOD Conference*, Boston, MA., June 1984, 1-8.
11. D. J. DeWitt and R. H. Gerber, ''Multiprocessor Hash-Based Join Algorithms'', *Proceedings of the Conference on Very Large Data Bases*, Stockholm, Sweden, August 1985, 151-164.
12. D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar and M. Muralikrishna, ''GAMMA - A High Performance Dataflow Database Machine'', *Proceedings of the Conference on Very Large Data Bases*, Kyoto, Japan, August 1986, 228-237.
13. D. J. DeWitt, S. Ghandeharizadeh and D. Schneider, ''A Performance Analysis of the GAMMA Database Machine'', *Proceedings of the ACM SIGMOD Conference*, Chicago, IL., June 1988, 350-360.
14. D. J. DeWitt, ''The Wisconsin Benchmark: Past, Present, and Future'', in *Database and Transaction Processing Systems Performance Handbook*, J. Gray (editor), Morgan-Kaufman, San Mateo, CA, 1991.
15. R. Epstein, ''Techniques for Processing of Aggregates in Relational Database Systems'', *UCB/Electronics Research Lab. Memorandum M79/8* (February 1979), University of California.
16. R. Fagin, J. Nievergelt, N. Pippenger and H. R. Strong, ''Extendible Hashing: A Fast Access Method for Dynamic Files'', *ACM Transactions on Database Systems 4*, 3 (September 1979), 315-344.
17. S. Fushimi, M. Kitsuregawa and H. Tanaka, ''An Overview of The System Software of A Parallel Relational Database Machine GRACE'', *Proceeding of the Conference on Very Large Data Bases*, Kyoto, Japan, August 1986, 209-219.
18. G. Graefe, ''Encapsulation of Parallelism in the Volcano Query Processing System'', *Proceedings of the ACM SIGMOD Conference*, Atlantic City, NJ., May 1990, 102.
19. G. Graefe, ''Parallel External Sorting in Volcano'', *submitted for publication, also CU Boulder Comp. Sci. Tech. Rep. 459*, February 1990.
20. G. Graefe and S. S. Thakkar, ''Tuning a Parallel Database Algorithm on a Shared-Memory Multiprocessor'', *CU Boulder Comp. Sci. Tech. Rep. 470*, April 1990.
21. G. Graefe and D. L. Davison, ''Architecture-Independent Parallel Query Evaluation in Volcano'', *submitted for publication, also CU Boulder Comp. Sci. Tech. Rep. 500*, December 1990.
22. G. Graefe, ''Volcano, An Extensible and Parallel Dataflow Query Processing System'', *accepted for publication in IEEE Transactions on Knowledge and Data Engineering*, . A more detailed version is available as CU Boulder Computer Science Technical Report 481, July 1990.
23. L. M. Haas, W. F. Cody, J. C. Freytag, G. Lapis, B. G. Lindsay, G. M. Lohman, K. Ono and H. Pirahesh, ''An Extensible Processor for an Extended Relational Query Language'', *Computer Science Research Report*, San Jose, CA., April 1988.
24. B. R. Iyer and D. M. Dias, ''System Issues in Parallel Sorting for Database Systems'', *Proceedings of the IEEE Conference on Data Engineering*, Los Angelos, CA, February 1990, 246.
25. 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.
26. M. Kitsuregawa, H. Tanaka and T. Motooka, ''Application of Hash to Data Base Machine and Its Architecture'', *New Generation Computing 1*, 1 (1983).
27. M. Kitsuregawa, M. Nakayama and M. Tagaki, ''The effect of bucket size tuning in the dynamic hybrid GRACE hash join method'', *Fifteenth International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, 1989, 257.
28. A. Klug, ''Access Paths in the 'ABE' Statistical Query Facility'', *Proceedings of the ACM SIGMOD Conference*, Orlando, FL., June 1982, 161-173.
29. D. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison-Wesley, Reading, MA., 1973.
30. R. P. Kooi, ''The Optimization of Queries in Relational Databases'', *Ph.D. Thesis*, September 1980.
31. R. A. Lorie and H. C. Young, ''A low communication sort algorithm for a parallel database machine'', *Fifteenth International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, 1989, 125.
32. J. Menon, ''A Study of Sort Algorithms for Multiprocessor Database Machines'', *Proceeding of the Conference on Very Large Data Bases*, Kyoto, Japan, August 1986, 197-206.
33. J. Ousterhout, ''Why Aren't Operating Systems Getting Faster as Fast as Hardware?'', *Dec. WRL Technical Note TN-11*, Palo Alto, CA., October 1989.
34. J. E. Richardson and M. J. Carey, ''Programming Constructs for Database System Implementation in EXODUS'', *Proceedings of the ACM SIGMOD Conference*, San Francisco, CA., May 1987, 208-219.

35. B. Salzberg, ''Merging Sorted Runs Using Large Main Memory'', *Acta Informatica 27* (1990), 195-215, Springer International.
36. B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren and B. Vaughan, ''FastSort: An Distributed Single-Input Single-Output External Sort'', *Proceedings of the ACM SIGMOD Conference*, Atlantic City, NJ., May 1990, 94.
37. D. Schneider and D. DeWitt, ''A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment'', *Proceedings of the ACM SIGMOD Conference*, Portland, OR, May-June 1989, 110.
38. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, ''Access Path Selection in a Relational Database Management System'', *Proceedings of the ACM SIGMOD Conference*, Boston, MA., May-June 1979, 23-34.
39. L. D. Shapiro, ''Join Processing in Database Systems with Large Main Memories'', *ACM Transactions on Database Systems 11*, 3 (September 1986), 239-264.
40. M. Stonebraker, E. Wong, P. Kreps and G. D. Held, ''The Design and Implementation of INGRES'', *ACM Transactions on Database Systems 1*, 3 (September 1976), 189-222.
41. Teradata, *DBC/1012 Data Base Computer, Concepts and Facilities*, Teradata Corporation, Los Angeles, CA., 1983.
42. S. Todd, ''PRTV: An efficient implementation for large relational data bases'', *Proceedings of the Conference on Very Large Data Bases*, 1975, 554-556.
43. H. Zeller and J. Gray, ''An Adaptive Hash Join Algorithm for Multiuser Environments'', *Sixteenth International Conference on Very Large Data Bases*, Brisbane, Australia, 1990, 186.
44. G. K. Zipf, *Human Behavior and the Principle of Least Effort, an Introduction to Human Ecology*, Addison-Wesley, Reading, MA., 1949.