Sort vs. Hash Revisited

Goetz Graefe, Ann Linville, and Leonard D. Shapiro

Abstract— **Efficient algorithms for processing large volumes of data are very important both for relational and new object-oriented database systems. Many query-processing operations can be implemented using sort- or hash-based algorithms, e.g., intersection, join, 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 often considered generally superior to sort-based algorithms such as merge-join. In this article, we compare the concepts behind sort- and hash-based query-processing algorithms and conclude that 1) many dualities exist between the two types of algorithms, 2) their costs differ mostly by percentages rather than factors, 3) several special cases exist that favor one or the other choice, and 4) there is a strong reason why both hash- and sort-based algorithms should be available in a query-processing system. Our conclusions are supported by experiments performed using the Volcano query execution engine.**

Index Terms—**Database query processing, value-matching, performance, sorting, merge-join, hashing, hash join, hybrid hash join, comparison, duality**

I. 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. For today's emerging database systems and their projected applications, algorithms for manipulating large data volumes remain very important because they are the key to providing acceptable performance not only for traditional value matching such as relational joins but also for manipulation of large set-valued attributes, maintenaince of some access paths such as access support relations [27], and data reduction in statistics and decision support.

In early relational research and implementation efforts, e.g., Ingres [16], [32], [44], System R [2], [6], PRTV [46], and ABE [30], 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 [42].

Although set processing was based on sorting, even early systems employed hash-based algorithms and data structures in the form of hash indices [44]. 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 [18], [28] and Gamma [10], [12], but also for sequential query execution engines [8], [43]. 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 where 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 major database system vendors are incorporating hash join and aggregation into their products, e.g., Tandem [47]. 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 algorithms, as the Teradata machine proves [45]. In fact, the choices of partitioning and local processing methods are independent or orthogonal from one another.

In this article, we compare sort- and hash-based algorithms, and argue, contrary to current "wisdom," as follows.

1) Many dualities exist between the two types of algorithms.
2) Their costs differ mostly by percentages rather than factors.
3) Many special cases exist that favor one or the other choice.
4) There is a strong reason why both hash- and sort-based algorithms should be available in a query-processing system.

The remainder of this article is organized as follows. We discuss sort- and hash-based algorithms as used in real systems or proposed in the literature in Section II. In Section III, we consider dualities and differences between sort- and hash-based query-processing algorithms. An experimental comparative study of sort- and hash-based algorithms follows in Section IV, using relational join as a representative for binary set matching algorithms. Section V contains a summary and our conclusions.

II. 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.

Manuscript received July 1991; revised December 1991. This work was supported in part by the National Science Foundation under Grants IRI-8996270, IRI-8912618, and IRI-9006348, and in part by the Oregon Advanced Computing Institute (OACIS), ADP, Intel Supercomputer Systems Division, and Sequent Computer Systems.

G. Graefe is with Microsoft Corp., Redmond, WA USA.

L. D. Shapiro is with the Department of Computer Science, Portland State University, Portland, OR 97207–0751 USA.

A. Linville is with the Department of Computer Science, University of Colorado at Boulder, CO 80309–0430 USA.

IEEE Log Number 9213324.

1041–4347/94$04.00 © 1994 IEEE

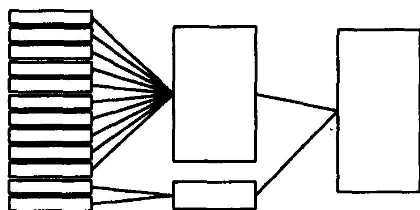

Fig. 1. Naïve merging.

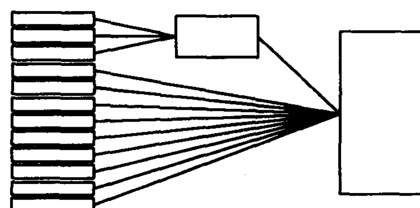

Fig. 2. Optimized merging.

Significant effort has been spent on devising and improving sort algorithms for database systems; recent work includes [1], [39]. 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*, i.e., the number of runs merged simultaneously. 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 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 [21], [39].

Another important optimization for sorting concerns the merge strategy. Let us explain it with an example shown in Fig. 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, and then merge first three of the 12 runs, and finally to merge the output with the remaining nine runs, as shown in Fig. 2. The I-O cost (measured by the number of memory loads that must be written to disk for all of the runs created) for the first strategy is $12 + 10 + 2 = 24$, wheeas 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 available for merging. More detailed examples are given in [21].

Recently, parallel sorting has found increased interest, e.g., in [3], [5], [15], [21], [23], [25], [33], [34], [40]. 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 article, we do not concern ourselves much with parallelism, because we believe that the issues of data manipulation and parallelism can be made orthogonal [19], [22], and that our conclusions are directly applicable to algorithms used in parallel environments.

For duplicate elimination and aggregate functions, e.g., a sum of salaries by department, Epstein's work has led to the use of sorting for aggregation, too [16]. 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 1000 groups using a sort with a maximal fan-in of 10. If aggregation is done separately from sorting; i.e., after sorting is complete, the largest run file may contain 10 000 records. If aggregation is done early, the largest run file will contain at most 1000 records. 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], [28], predominantly for relational join. Because they were used in a number of relational database machines, hash-based join algorithms were frequently identified with parallel query execution [11], [12], [18], even though they make equal sense in sequential environments. In its simplest form, called *classic hash join* in [43], 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 [18] builds the overflow files before any overflow can occur. Bucket tuning and dynamic destaging can be used to optimize the performance of overflow avoidance [29], [35]. 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], [43] 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.

Fig. 3 shows how two inputs, say, R and S, are partitioned recursively in hash join algorithms. In practice, the files on

TABLE I
DUALITY OF SORT- AND HASH-BASED QUERY-PROCESSING ALGORITHMS

Aspect	Sorting	Hashing
In-memory algorithm	Quicksort	Classic Hash
Divide-and-conquer paradigm	Physical division, logical combination	Logical division, physical combination
Large inputs	Single-level merge	Partitioning into overflow files
I/O Pattern	Sequential write, random read Fan-in	Random write, sequential read Fan-out
I/O Optimization	Read-ahead, forecasting	Write-behind
Very large inputs	Multi-level merge Number of merge levels Non-optimal final fan-in	Recursive overflow resolution Recursion depth Non-optimal hash table size
Optimizations	Merge optimizations	Bucket tuning
Better use of memory	Reverse runs & LRU Replacement selection ?	Hybrid hash ? Single input in memory
Aggregation	Aggregation in replacement selection	Aggregation in hash table
Interesting orderings	Merge-Join without sorting	N-way joins, hash-merging

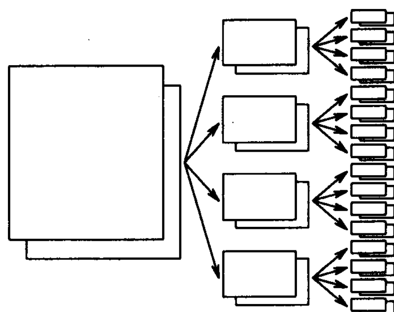

Fig. 3. Recursive hash join.

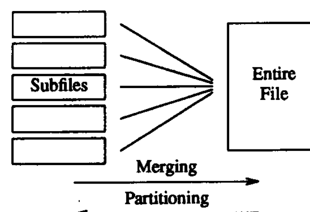

Fig. 4. Duality of partitioning and merging.

each level will not be of exactly equal size, depending on the data values and the hash function. If, in the deepest partitioning level, some of the R-outputs fit into the available memory, bucket tuning will choose which ones to keep in memory for immediate join processing while partitioning the S input. On the other hand, hybrid hash join will retain some of the R-files in memory without regard to their final size.

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

III. 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. When appropriate, however, we indicate specific operations.

Table I gives an overview of the features that correspond to one another. Both approaches permit in-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 a hashing technique. 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 *divide-and-conquer* paradigm employed for both sorting and hashing. This is an important similarity of sorting and hashing and has been observed before, e.g., by Bratbergsengen [8] and Salzberg [38]. 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. Fig. 4 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, 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 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*, because the limitation is encountered while writing partition files.

In order to keep the merge process active at all times, many merge implementations use read-ahead controlled by forecasting [31], trading reduced I-O delays for a reduced fan-in. The dual to read-ahead during merging is write-behind during partitioning, i.e., keeping a free output buffer that can be allocated to an output file while the previous page for that file is being written to a disk.

Considering the 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 repartitioned, the result repartitioned, and so forth, 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 come back to them later.

If merging is done in the most naïve 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 optima; i.e., it might not use the maximal possible fan-in. During hashing, 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 files on that level. Thus, in both approaches, the memory resources are not used optimally in the most naïve version of the algorithms.

In order to make best use of the final merge (which, by definition, includes all output items), this merge 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. Using memory more effectively and using less than the maximal fan-out in hashing has been addressed in research on bucket tuning [29].

The development of hybrid hash algorithms [10], [43] 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 to 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 [31]. If the runs' sizes are doubled, their number is cut in half. Therefore, merging can be reduced to some amount, namely, by $\log_F(2) = 1/\log_2(F)$ merge levels where F is the fan-in of the merge, i.e., the number of run files that can be combined in a single step. Note that if the fan-in F is large, the effect of replacement selection and larger runs on the merge depth and the total merge effort is negligible. However, if two sort operations feed into a merge-join and both final merges are interleaved with the join, each 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.

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 or 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 that achieved with classic hash is easy to achieve.

The final entry in Table I concerns *interesting orderings* used in the System R query optimizer [42], and presumably other query optimizers as well. 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 sort operators between merge-join operators. For joining three relations, as shown in Fig. 5, 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 $2N2$ for joins on different attributes.

Hash-based algorithms tend to produce their outputs in a very unpredictable order. To take advantage of multiple

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

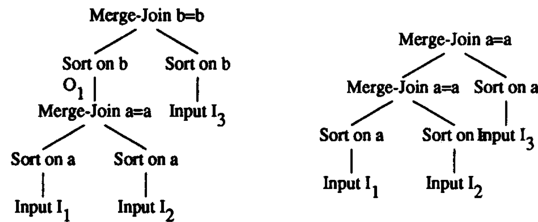

Fig. 5. The effect of interesting orderings.

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 cumbersome to implement, in particular if some of the "join" operations can actually be semijoin, 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. Although it has not been reported explicitly in the literature, it is perfectly possible to implement a join algorithm that uses two hash indices like merge-join uses two B-trees, provided that the same hash function was used to create the indices. For example, it is easy to imagine "merging" the leaves (data pages) of two extendable hash indices [17], even if the key cardinalities and distributions are very different.

In summary, there exist many dualities between sorting using multilevel 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.

IV. Experimental Comparison of Sorting and Hashing

In this section, we report on a number of experiments to demonstrate that the duality of sorting and hashing leads to similar performance in many cases, to illustrate 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. We have chosen relational join as a representative of binary set matching algorithms because it is a very frequently used database operation, and because many fundamental operations useful in all database systems, e.g., intersection, union, and difference, can all be realized with sort- and hash-based join algorithms. We first describe the experimental environment and then report on a series of experiments.

A. Experimental Environment

The test bed for our experiments was the Volcano extensible and parallel query-processing engine [22]. The purpose of the Volcano project is to provide efficient, extensible tools for query and request processing in novel application domains, particularly in object-oriented and scientific database systems, and for experimental database performance research. 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 algebraic 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.

Since almost all queries require more than one operator, state records can be linked 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 quadruple of 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 topmost 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 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 model used in relational products, e.g., DB2, Ingres, Informix, and Oracle, but also the iterator concept in the E database language [37] and the algebraic query evaluation system of the Starburst extensible-relational database system [24]. Table II gives algorithm outlines for some operators' *open*, *next*, and *close* procedures.

Fig. 6 shows a simple query plan that 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 opens the left file scan, requests all records from the file scan by calling its *next* function, and closes it. After calling *open* on the right file scan, the hash join operator is ready to produce data. Its *open*

TABLE II
EXAMPLES OF ITERATOR FUNCTIONS

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* left "build" input; build hash table calling *next* on build input; *close* build input; *open* right "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 only one merge step is left; open the remaining run files	determine next output item; read new item from the correct run file	destroy remaining run files

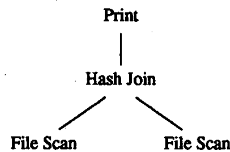

Fig. 6. A simple query plan.

procedure terminates, and print's *open* procedure returns to the driver module.

Now the entire 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* function returns. The query execution driver calls the topmost operator's *next* function repeatedly until it receives an error status. When, in a subsequent *next* call, the right file scan returns an end-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, semijoin, outer join, antijoin, union, intersection, difference, antidifference, aggregation, and duplicate elimination [26]. Volcano includes both a sort- and a hash-based implementation of the one-to-one match operator. The sort-based version combines a sort operator that includes aggregation and duplicate elimination [21] with a generalized merge-join operator. The hash-based version is a recursive implementation of hybrid hash join hash augmented with aggregation during the build phase and parameterized to allow both overflow avoidance similar to Grace hash join [18] and overflow resolution as the original hybrid hash join [10], [43]. 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, even though replacement selection 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. Because 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 heap size 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 we abandoned the idea of using heaps for creating initial runs. Furthermore, replacement selection with copying into a holding area does not work easily for variable-length records.

Volcano is operational on a variety of UNIX machines, including several parallel systems [19], [20]. 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, and so forth, and the other was accessed directly by Volcano as a raw device.

[2]Note that 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 [36]. In a shared-memory parallel machine in which bus bandwidth may be scarce, avoiding copying is even more important.

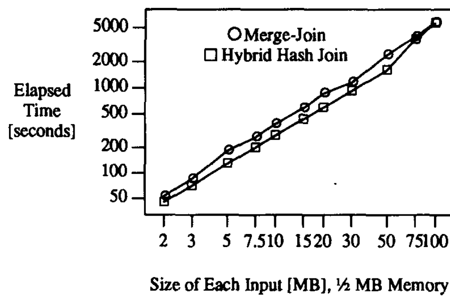

Fig. 7. Join performance for equal input sizes.

B. Joins with Equal Input Sizes

In order to demonstrate the relative performance of sorting and merge 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 4-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 1/2 megabyte. The cluster size (unit of I-O) was 4 kilobytes.

Fig. 7 shows the performance for merge and hybrid hash join for input sizes between about 2 megabytes (10 000 tuples) and about 100 megabytes (500 000 tuples). Sort and merge-join performance is indicated with circles (◯), hybrid hash with squares (□). Note that both axes are logarithmic. The performance is not exactly linear with the input sizes, because both algorithms, merge and hybrid hash join, require multiple levels of merging or overflow resolution for the larger inputs.

The difference between merge-join and hybrid hash join is small, certainly far from an order of magnitude. The difference in the performance of sort- and hash-based joins stems from the fact that sorting requires both inputs in memory, whereas hashing "filters" the second input through the hash table, which contains only items from the first input. As expected from the discussion in the section on duality, this disadvantage of sorting could be offset by using replacement selection for creating initial sorted runs. To verify this claim for the concrete example, we calculate the relative I-O required for sorting using quicksort, sorting using replacement selection, and hybrid hash to join two 50-megabyte inputs. We calculate write costs for only one input because the I-O is equal for both inputs, and all files written will be read exactly once. Using quicksort, 50 megabytes of data divided by 1/2 megabyte of memory results in 100 runs. Because each sort can use a final merge fan-in 64(1/2 megabytes/4 megabytes/2), 100 runs must be reduced to 64 by using a fan-in of 127 (1/2 megabytes/4 kilobytes -1 requiring 37 $(100-64+1)$ original runs to be merged into one 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 megabytes (50 megabytes/127). 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.

We would like to discuss why we have obtained different results than Schneider and DeWitt [41] and Shapiro [43]. First, Schneider and DeWitt joined two relations with different sizes (about 2 megabytes and 20 megabytes). Later we come back to join inputs of different size. A second 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's [9]; i.e., it sorts from a disk-resident file into a disk-resident file. Therefore, an intermediate result must be written to disk before it can be sorted, rather than being sorted into initial runs before the first write step, and the entire sorted file is written back to disk rather than being 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 [41] is biased against sort-based algorithms. Shapiro [43] 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 multilevel recursion or merging are not common. This argument does not always hold, however, as discussed in the next section.

C. 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 naïve overflow avoidance (write all partitions to disk, do not retain some data in memory) is that as much data as possible can be kept in memory; i.e., it is never written to temporary files. This led us to search for a dual in the realm of sorting. To achieve the same effect, we changed Volcano's sort operator so that it retains data in memory from the last quicksort until the first merge. In order to achieve that, it writes runs in reverse, i.e., in descending order for an ascending sort, and for the clusters written after the end of the input has been found, it gives a hint to the buffer manager to ensure that those clusters are replaced in a LRU discipline. As many clusters as possible will remain 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.

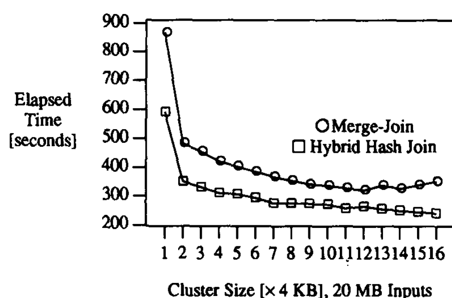

Fig. 8. Join performance by cluster size.

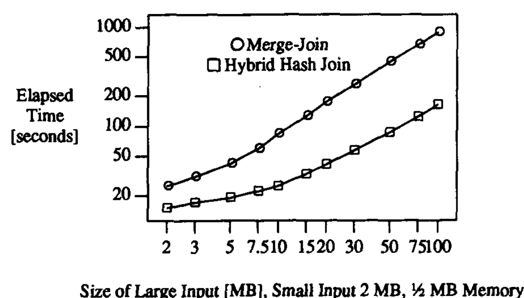

Fig. 9. Join performance for different input sizes.

This optimization has been analyzed in some studies, e.g., [43], 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 naïve 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 [21] beyond the effect of read-ahead and double buffering [39]. 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 they 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 moderate merge fan-ins and relatively large clusters. If merging and partitioning are indeed duals, the same effect of cluster size on hybrid hash performance can be expected.

Fig. 8 shows the performance of joins of two 20-megabyte inputs for various cluster sizes. As can be seen, hash performance is as sensitive to cluster size as sorting. A similar effect was considered 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 or 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 (when one ignores the effects of rounding in the precise cost function) depends only on the memory size, not on the input sizes [21]. The same is true for hashing. Exploiting a proven sort optimization for hash-based algorithms is the second optimization we transferred, based on our duality considerations. In the following experiments, we used clusters of 32 kilobytes and fan-ins and fan-outs of 15.

D. 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 so that each tuple in the smaller relation has exactly one match in the larger relation. For sorting input of a merge-join, each input determines the number of merge levels. 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, so 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. Reversing the roles of build and probe inputs dynamically, e.g., after a first partitioning step, is possible, but is not considered further in this article.

Fig. 9 shows the performance of merge- and hybrid hash join for input of equal to very different size. The smaller (build) input size is fixed at 2 megabytes, and the larger (probe) input size varies between 2 megabytes and 100 megabytes. As can be seen, the performance advantage of hybrid hash join increases with the size difference between the input relations. The reason is that for hybrid hash join, $1/4$ of the build relation fits into memory and $3/4$ 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 for semijoin, outer join, intersection, union, and difference derived from merge- and hybrid hash join will perform very differently for inputs of different sizes. On the other hand, if the query optimizer cannot reliably predict which input is smaller, merge-join may be the superior choice.

E. Joins with Skewed Data and Hash Value Distributions

Finally, we experimented with some skewed join value distributions. Instead of using a uniform random number

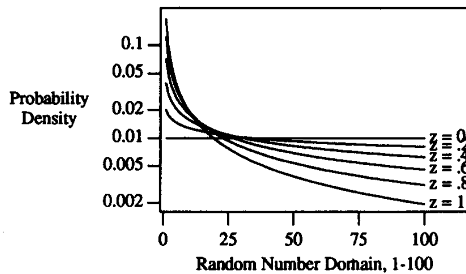

Fig. 10. Probability distributions for selected values of z.

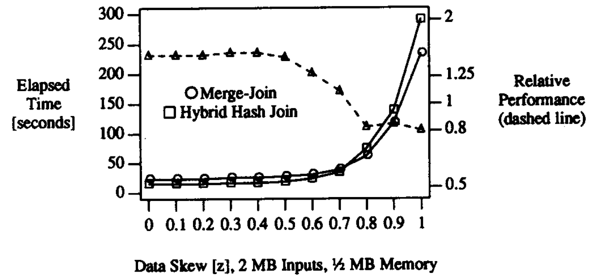

Fig. 11. Join performance for skewed data.

generator to create test data, we used a generalized random function borrowed from Knuth [31]. Using a continuous parameter, z probabilities are assigned to the numbers 1 to N as $P_i = 1/i^z/c$ for $i = 1, \cdots, N$, where $c = \sum_{j=1}^{N} 1/j^z$ is a normalization factor used to ensure that the sum of all probabilities is 1. For $z = 0$, this random function creates uniform data; for $z = 1$, the function can be used to create random data according to Zipf's law [48]. The reason why Zipfian distributions are relevant for our purpose is that they were defined to model real data and their frequencies.

Fig. 10 shows the probability of values $N = 1, \cdots, 100$ with $z = i/5$ for $i = 0, \cdots, 5$. Since the domain of N is discrete, 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 smaller than for $N = 1$ following Zipf's law. Probabilities with more skew can be obtained with higher values of z.

We used the same skewed data distribution in both inputs. Compared to uniform distributions of join keys 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.

Fig. 11 shows the effect of skew on the performance of merge- and hybrid hash join, including the relative performance of merge- and hybrid hash join under skew. It is evident that merge-join is less affected by the skew. For uniform data, hybrid hash join outperforms merge-join, as shown in the previous figures. For highly skewed data, sorting and merge-join outperforms hybrid hash join. The reason is that the partitioning is not even; for $z = 1$, a large fraction of the build and probe inputs (3/4 of their data items) is written to one pair of overflow files. 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 built according to memory size, and an input record is 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 are against the idea of hashing, i.e., randomizing, the data. To counteract and possibly even exploit hash value skew, we are working on using hash value skew to assign hash values to the in-memory hash table and to partition files, applying overflow avoidance, hybrid hash, or nested loops join for partition files as appropriate.

V. Conclusion

In this paper, we have outlined many dualities between sort- and hash-based query processing algorithms, e.g., for intersection, join, and duplicate elimination. Under many circumstances, the cost differs by percentages rather than by factors, presuming that the algorithms have been implemented and tuned with similar care. We expected this result from the large number of dualities and verified it with the Volcano query-processing system.

Two special cases exist that favor one or the other, however. First, if the inputs of a binary operator are of very different size (and the query optimizer can reliably predict this difference), hash-based algorithms will outperform sort-based algorithms, because only the smaller of the two inputs determines how many recursion levels are required or what fraction of the input files must be written to temporary disk files during partitioning whereas 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. Second, if the hash value distribution is not uniform, hash partitioning performs very poorly and creates significantly higher costs than sort-based methods do. 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 be chosen by the query optimizer according to the two cases above. If both cases arise simultaneously, i.e., if a join of differently sized inputs with unpredictable hash value distribution, the query optimizer must estimate which

one poses the greater danger to system performance and predictability, and must choose accordingly.

The important conclusion from this research is that neither the input size nor the memory size determines 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, by the danger of nonuniform hash value distributions, and by the opportunities to exploit interesting orderings. Furthermore, because neither algorithm type outperforms the other in all situations, and because realistic situations exist that favor one or the other, both should be available in a query execution engine for a choice to be made in each case by the query optimizer.

ACKNOWLEDGMENT

The initial interest in comparing sort- and hash-based algorithms in greater detail resulted from a spirited discussion with B. Lindsay and H. Pirahesh during the VLDB Conference in 1988. D. DeWitt, D. Schneider, and the anonymous reviewers made insightful comments on earlier drafts.

REFERENCES

[1] A. Aggarval and J. S. Vitter, "The input/output complexity of sorting and related problems," *Commun. ACM* vol. 31, p. 1116, Oct. 1988.

[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 Trans. Database Syst.* vol. 1, no. 2, p. 97, June 1976 (reprinted in M. Stonebraker, *Readings in Database Systems.* San Mateo, CA: Morgan Kaufmann, 1988

[3] M. Beck, D. Bitton, and W. K. Wilkinson, "Sorting large files on a backend multiprocessor," *IEEE Trans. Comput.,* vol. 37, p. 769, 1988.

[4] D. Bitton and D. J. DeWitt, "Duplicate record elimination in large data files," *ACM Trans. Database Syst.*, vol. 8, no. 2, **p. 255**——, June 1983.

[5] D. Bitton Friedland, "Design, analysis, and implementation of parallel external sorting algorithms," Comput. Sci. Tech. Rep. 464, University of Wisconsin—Madison, Jan. 1982.

[6] M. Blasgen and K. Eswaran, "On the evaluation of queries in a relational database system," IBM Res. Rep. RJ-1745, San Jose, CA, USA, Apr. 8, 1976.

[7] ——, "Storage and access in relational databases," *IBM Syst. J.*, vol. 16, no. 4, 1977.

[8] K. Bratbergsengen, "Hashing methods and relational algebra operations," *Proc. Int. Conf. Very Large Data Bases*, 1984, p. 323.

[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,* vol. 15, no. 10, p. 943, Oct. 1985.

[10] 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.*, 1984, p. 1.

[11] D. J. DeWitt and R. H. Gerber, "Multiprocessor hash-based join algorithms," *Proc. Int. Conf. Very Large Data Bases*, 1985, p. 151.

[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," *Proc. Int. Conf. Very Large Data Bases*, 1986, p. 228 (reprinted in M. Stonebraker, *Readings in Database Systems.* San Mateo, CA: Morgan Kaufmann, 1988).

[13] D. J. DeWitt, S. Ghandeharizadeh, and D. Schneider "A performance analysis of the GAMMA database machine," *Proc. ACM SIGMOD Conf.*, 1988, p. 350.

[14] D. J. DeWitt, "The Wisconsin benchmark: Past, present, and future," in J. Gray, Ed., *Database and Transactions Processing Systems Performance Handbook.* San Mateo, CA: Morgan Kaufmann, 1991.

[15] D. J. Dewitt, J. Naughton, and D. Schneider, "Parallel sorting on a shared-nothing architecture using probabilistic splitting," *Proc. Int. Conf. Parallel Distrib. Inform. Syst.*, Miami Beach, FL, USA, Dec. 1991.

[16] R. Epstein, "Techniques for processing of aggregates in relational database systems," UCB/Electron. Res. Lab. Memo. M79/8, Univ. of California, Feb. 1979.

[17] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing: A fast access method for dynamic files," *ACM Trans. Database Syst.*, vol. 4, no. 3, p. 315——, Sept. 1979.

[18] S. Fushimi, M. Kitsuregawa, and H. Tanaka, "An overview of the system software of a parallel relational database machine GRACE," *Proc. Int. Conf. Very Large Data Bases*, Kyoto, Japan, Aug. 1986.

[19] G. Graefe, "Encapsulation of parallelism in the Volcano query processing system," *Proc. ACM SIGMOD Conf.*, 1990, p. 102——.

[20] G. Graefe and D. L. Davison, "Encapsulation of parallelism architecture-independence in extensible database query processing," *IEEE TRans. Software Eng.*, vol. 19, no. 8, pp. 747, Aug. 1993.

[21] G. Graefe, "Parallel external sorting in Volcano," Tech. Rep. 459, Univ. of Colorado, Boulder, USA, Dept. Comput. Sci., 1991.

[22] ——, "Volcano: An extensible and parallel dataflow query processing system," *IEEE Trans. Knowledge. Data Eng.*, vol. 6, no. 1, pp. 120–135, Feb. 1994.

[23] G. Graefe and S. S. Thakkar, "Tuning a parallel database algorithm on a shared-memory multiprocessor," *Software—Practice and Experience*, vol. 22, no. 7, p.495, July 1992.

[24] 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," Comput. Sci. Res. Rep., San Jose, CA, USA, Apr. 1988.

[25] B. R. Iyer and D. M. Dias, "System issues in parallel sorting for database systems," *Proc. IEEE Conf. Data Eng.* 1990, p. 246.

[26] T. Keller and G. Graefe, "The one-to-one match operator of the Volcano query processing system," Oregon Graduate Center, Comput. Sci. Tech. Rep., Beaverton, OR, USA, June 1989.

[27] A. Kemper and G. Moerkotte, "Access support in object bases," *Proc. ACM SIGMOD Conf.*, 1990, p. 364.

[28] M. Kitsuregawa, H. Tanaka, and T. Motooka, "Application of hash to data base machine and its architecture," *New Generation Computing*, vol. 1, 1983.

[29] A. M. Kitsuregawa, M. Nakayama, and M. Takagi, "The effect of bucket size tuning in the dynamic hybrid GRACE hash join method," *Proc. Int. Conf. Very Large Data Bases*, 1989, p. 257.

[30] A. Klug, "Access paths in the 'ABE' statistical query facility," *Proc. ACM SIGMOD Conf.*, 1982, p. 161.

[31] D. Knuth, *The Art of Computer Programming: Sorting and Searching* , vol. III. Reading, MA: Addison-Wesley, 1973

[32] R. P. Kooi, "The optimization of queries in relational databases," Ph.D. dissertation, Case Western Reserve Univ., OH, USA, Sept. 1980.

[33] R. A. Lorie and H. C. Young, "A low communication sort algorithm for a parallel database machine," *Proc. Int. Conf. Very Large Data Bases*, 1989, p. 125.

[34] J. Menon, "A study of sort algorithms for multiprocessor database machines," *Proc. Int. Conf. Very Large Data Bases*, 1986, p. 197.

[35] M. Nakayama, M. Kitsuregawa, and M. Takagi, "Hash-partitioned join method using dynamic destaging strategy," *Proc. Int. Conf. Very Large Data Bases*, 1988, p. 468.

[36] J. Ousterhout, "Why aren't operating systems getting faster as fast as hardware?" WRL Tech. Rep. TN-11, Palo Alto, CA, USA, Oct. 1989.

[37] J. E. Richardson and M. J. Carey, "Programming constructs for database system implementation in EXODUS," *Proc. ACM SIGMOD Conf.*, 1987, p. 208.

[38] B. Salzberg, *File Structures: An Analytic Approach.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

[39] ——, "Merging sorted runs using large main memory," *Acta Informatica*, vol. 27, p. 195, 1990.

[40] B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren, and B. Vaughan "FastSort: A distributed single-input single-output external sort," *Proc. ACM SIGMOD Conf.*, 1990, p. 94.

[41] D. Schneider and D. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," *Proc. ACM SIGMOD Conf.*, 1989, p. 110.

[42] 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.*, 1979, p. 23 (reprinted in M. Stonebraker, *Readings in Database Systems.* San Mateo, CA: Morgan-Kaufman, 1988).

[43] L. D. Shapiro, "Join processing in database systems with large main memories," *ACM Trans. Database Syst.*, vol. 11, no. 3, p. 239, Sept. 1986.

[44] M. Stonebraker, E. Wong, P. Kreps, and G. D. Held, "The Design and Implementation of INGRES," *ACM Trans. Database Syst.*, vol. 1, no. 3, p. 189, Sept. 1976 (reprinted in M. Stonebraker, *Readings in Database Systems*. San Mateo, CA: Morgan-Kaufman, 1988).
[45] Teradata Corp., DBC/1012, *Data Base Computer, Concepts, and Facilities*, Los Angeles, CA, USA, 1983.
[46] S. Todd, "PRTV: An efficient implementation for large relational data bases," *Proc. Int. Conf. Very Large Data Bases*, 1975, p. 554.
[47] H. Zeller and J. Gray, "an adaptive hash join algorithm for multiuser environments," *Proc. Int. Conf. Very Large Data Bases*, Brisbane, Australia, 1990.
[48] G. K. Zipf, *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Reading, MA: Addison-Wesley, 1949.

G. Graefe was an undergraduate student in business administration and computer science in Germany before he received the M.S. and Ph.D. degrees in computer science from the University of Wisconsin—Madison, in 1984 and 1987, respectively.

In 1987, he joined the faculty of the Oregon Graduate Institute, where he initiated both the Volcano project on extensible query processing and, with David Maier, the REVELATION project on OODB performance. From 1989 to 1994, he was an Assistant Professor of Computer Science at the University of Colorado at Boulder. He is currently working on extensions to Volcano, including a new optimizer generator, request processing in object-oriented and scientific database systems, optimization and execution of very complex queries, and physical database design. His thesis work at the University of Wisconsin was the EXODUS Optimizer Generator.

A. Linville received the B.S. degree in geology from Florida Atlantic University, Boca Raton, FL, USA.

She worked in the oil industry for several years before returning to school. She is currently a graduate student in computer science at the University of Colorado at Boulder. She has worked for the past year on the Volcano extensible query-processing system.

L. D. Shapiro received the B.A. degree in mathematics from Reed College, Portland, OR, USA, in 1965, and the Ph.D. degree in mathematics from Yale University, New Haven, CT, USA, in 1969.

Currently, he is a Professor and Chair of Computer Science at Portland State University, Portland, OR, USA. Previously, he was a member of the faculty at North Dakota State University, Fargo, ND, USA, and the University of Minnesota, Minneapolis. He has served as an investigator on several research projects funded by, among others, the National Science Foundation, the U.S. Department of Health, Education, and Welfare, the U.S. Air Force Office of Scientific Research, and the U.S. Department of Agriculture. In addition, he has done extensive consulting work for local and national businesses and industries. His current research interests are in database management systems performance issues.

Dr. Shapiro is a member of ACM and the IEEE Computer Society.