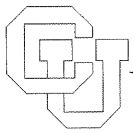
Heap-Filter Merge Join:
A New Algorithm for Joining Medium-Size Inputs

Goetz Graef

CU-CS-471-90 May 1990



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Heap-Filter Merge Join:
A New Algorithm for Joining Medium-Size Inputs

Goetz Graefe

University of Colorado at Boulder
graefe@boulder.colorado.edu

Abstract

We present a new algorithm for relational equi-join. The algorithm is a modification of merge join but promises superior performance for medium-size inputs. In many cases, it even compares favorably with hybrid hash join. We present I/O cost comparisons for a sequential implementation but also discuss parallel versions of the new algorithm.

1. Introduction

One of the most expensive operations in relational database systems is the join which matches tuples (rows, records) from two relations (tables, files) with equal key values. While there seems to be an abundance of relational join algorithms, e.g. [1-6, 9, 12, 13, 16, 17, 19-29, 31-34, 36-40], some cases still allow improvement. In this brief report, we present a new algorithm which is a variant of the well-known merge join algorithm. In essence, it avoids merging sorted runs of the outer input and merges them directly with the inner input. This method avoids a substantial amount of I/O for sorting the outer input at the expense of increased I/O for the inner input.

In the following section, we discuss two join algorithms that are commonly regarded as very effective, namely merge join and hybrid hash join, and derive their cost formulas. The new algorithm, called *heap-filter merge join*, is described in Section 3. In Section 4, we provide an analytical performance comparison of merge join, hybrid hash join, and heap-filter merge join. Section 5 contains a summary and our conclusions from this effort.

2. Competitive Join Algorithms

In this section, we discuss two known join algorithms and their cost formulas. We have chosen merge join because heap-filter merge join is a derivative of it, and hybrid hash join because it was shown to be a very efficient join algorithm [8]. Please note that in all our cost formulas, we omit the cost of creating unsorted streams of input tuples and any cost associated

with storing the output tuples since these costs are common to and equal for all the algorithms.

2.1. Merge Join

Merge join has been one of the first join algorithms to be published and analyzed, e.g. in [2,3,33,37], and is the algorithm of choice for large inputs in almost all commercial database systems. The idea is quite simple and well-known: Sort both inputs on the join attribute[1], and merge them advancing a scan pointer in each input. If both inputs contain duplicate join attribute values, the scan pointer in the inner input sometimes has to be backed up.

The major cost is created by the sort steps. For simplicity, we only concern ourselves with I/O costs, even though we realize that the CPU cost can be non-trivial. The I/O during sorting consists of sequential write operations while writing initial runs, and random read operations while reading and merging these runs. We assume that the memory size is a reasonable fraction of the input sizes; therefore we calculate the cost for only one merge level [30,31]. Using the cost parameters in Table 1, the cost for sorting the inner input is

$$sort\_inner := (seq + rnd)\ inner.$$

If we assume that the output of both sort operations is immediately passed to the merge join operator, i.e., without additional I/O, the total I/O cost for the merge join is

$$(seq + rnd)\ outer + sort\_inner$$

We will come back to this formula in Section 4.

inner	size of inner input in pages
outer	size of outer input in pages
memory	memory size in pages
seq	sequential I/O, 10ms
rnd	random I/O, 30ms

Table 1. Cost Components.

[1] We assume without loss of generality that there is only one join attribute. Our discussion is equally valid for multi-attribute equi-joins.

2.2. Hybrid Hash Join

Hashing is a very fast method for finding equality matches and a number of hash-based join algorithms have been proposed, e.g. [4-6, 12, 31]. A memory-resident hash table is built with the first input, called the *build input*, and then probed with the other input, called the *probe input.* This algorithm is simple and fast if the build input fits into main memory. A number of strategies have been proposed to deal with the case when the build input is larger than memory [5, 12, 31]. In a recent comparison of several algorithms, hybrid hash join was found to provide superior performance over a wide range of parameters [8].

Hybrid hash join is an optimistic hash join algorithm; we call it optimistic since it starts out with the assumption that the hash table will fit into memory and uses hash table overflow resolution when it becomes necessary. When hash table overflow occurs, some of the hash buckets are dumped from main memory to a *build overflow file* on disk. Further tuples from the build input are first checked whether they belong to a hash bucket still in memory or to one on disk; in the latter case, they are not kept in memory but immediately added to the overflow file. If the remaining hash buckets overflow again, more buckets are dumped, etc. Thus multiple overflow files can be created and added to while building the hash table. Notice that overflows are dealt with more efficiently if multiple overflow files are used [6].

After the build input is exhausted, the probe input is consumed. If a probe input tuple matches with a hash bucket in memory, the join is performed immediately. Otherwise, it is added to a *probe overflow file.* It makes good sense to build multiple probe overflow files using the same partitioning rule used for the build overflow files. After both inputs are consumed, the overflow files are joined using the same algorithm. For our analysis, we assume that a sufficient number of overflow files has been built such that no further overflow occurs, i.e., both inputs have been partitioned into small enough disjoint subsets.

The I/O cost for the overflow files depends on what fraction of these files has to be written to disk; we determine this fraction using the formula

$$F := (inner - memory) / inner$$

optimistically assuming a perfect hash function and partitioning. Writing to overflow files is

sequential if there is only one such file, otherwise it requires random writes. Reading overflow files always uses sequential I/O. Thus, if $inner > 2\ memory$, the I/O cost is

$$(rnd + seq)\ F\ (inner + outer),$$

otherwise, it is

$$(2\ seq)\ F\ (inner + outer)$$

3. Heap-Filter Merge Join

Merge join uses two sorted inputs and computes their (equi-) join by maintaining scan pointers in each, advancing them based on comparisons of join attributes and resetting them sometimes in the presence of duplicate join attribute values. The cost of the actual merge join algorithm is relatively small compared to the cost of sorting the inputs. Our effort was inspired by the desire to reduce the sorting costs.

We assume that the outer input is the larger of the two inputs. Classic hash join[2] performs very well and is hard to improve on if the inner input fits in main memory; therefore we will not concern ourselves with this case. Let us assume that the inner input's size is a moderate multiple of the memory size, say twice to ten times the size of memory, and that the outer input is quite large.

The new algorithm, which we call *heap-filter merge join*, avoids sorting the outer input completely (which is different from completely avoiding the sort!) and instead joins the initial sorted runs immediately with the inner input. Thus, such runs do not need to be written to disk or read for merging. The I/O savings compared to merge join are substantial – the outer input is never written to temporary files and therefore does not incur any I/O costs.

These saving, however, do not come without a cost, namely scanning the sorted inner input repeatedly. If *replacement selection* is used for run generation, i.e., if all tuples from the outer input have to travel through a sorting heap which gives this join algorithm its name, the number of runs can be expected to be the size of the outer input divided by twice the memory size [18]. Since the inner input must be joined with each of these runs, the inner input must be retrieved

[2] Classic hash join is hash join without overflow avoidance or resolution [31].

repeatedly from disk, once for each run of the outer input.

While this algorithm seems reminiscent of nested loops join and therefore very expensive, it does warrant a closer examination. Let us develop this algorithm's cost formula. First, we need to sort the inner input, at cost *sort_inner* developed above. Second, we need to scan the sorted inner input once for each run from the outer input. The number of these runs is equal to

$$R := outer \,/\, (2\, memory)$$

The cost for each scan over the inner input is

$$seq\ inner$$

Thus, the total I/O cost for heap-filter merge join is

$$R\ inner\ seq + sort\_inner$$

In his thesis, Kooi used a "sortedness factor" for the outer input of nested loops join to determine how effective block accesses to the inner input could be, i.e., how often a block of the inner input would be used consecutively [19]. In Kooi's terminology, heap-filter merge join uses a heap to enforce high "sortedness" of the outer input and to allow efficient access to the inner input.

3.1. Alternating Heap-Filter Merge Join

It would be desirable to leverage at least some of the I/O performed during a scan over the inner input for the next scan. This can easily be done by creating *alternating runs* from the outer input. This means that the first run is ascending, the next one descending, the third one ascending again, etc. For these runs, the sorted inner input can be scanned forward, backward, forward, backward, etc., using the last page of one scan as the first of the next without I/O. We call this algorithm *alternating heap-filter merge join.* Careful analysis will show that only one page should be used; using more memory pages will decrease the size of runs from the outer input and therefore increase the number of scans over the inner.

Unfortunately, when alternating runs are produced from the outer input, the average run size will decrease to 1½ times the size of memory, and more passes over the sorted inner input are

required[3]. The number of runs from the outer input, and therefore the number of scans of the sorted inner input, will be

$$R' := outer \;/\; (1½\ memory)$$

For alternating heap-filter merge join, the total I/O cost is

$$(R'\ (inner - 1) + 1)\ seq + sort\_inner$$

which will be higher than the cost of the heap-filter merge join discussed earlier if *inner* is greater than 3, i.e., alternating heap-filter merge join outperforms heap-filter merge join only when classic hash join outperforms either variant of heap-filter merge join.

3.2. Complex Queries

We would like to point out that heap-filter merge join not only has good performance as a single operator in a query (as we will see in the Section 4), it also allows dataflow between operators in a complex query. In particular, as soon as the first tuple from the outer input has traveled through the heap, it can be joined with the inner input and an output tuple can be produced. Note that all algorithms discussed here consume one input completely (for sorting or to build a hash table) before starting to consume the other input and to produce results.

3.3. Parallelism

Heap-filter merge join can also be used in parallel query evaluation systems. Let us consider three forms of parallelism, namely *pipelines*, *bushy-tree parallelism*, and *intra-operator parallelism* [14]. Pipelines can be exploited as discussed above, and therefore multiple sub-trees in a bushy query evaluation plan can be executed in parallel.

For intra-operator or data parallelism, there are basically two methods to parallelize join algorithms, *fragment-and-replicate*, e.g. [1,10-], and *partitioning* on the join attribute, e.g. [6,7], as well as mixed algorithms, e.g. [34]. Heap-filter merge join can be used with any of these. In fact, the parallelization strategy and the choice of join algorithm are orthogonal for equi-joins. Since we always assumed that the outer relation is the larger of the two input relations, it should be parti-

[3] If the memory size is M, and the independent probability that subsequent runs have opposite direction is p, the average run length is 6/(3+p) M; see exercise 5.4.1-24g on p. 265 in [18]. In alternating heap-filter merge join, this "independent" probability is 1.

tioned or fragmented and not be replicated. If the inner input is also partitioned, its partitions can easily be sorted in parallel [15], and we would expect linear speedup for all phases of the join. If the inner input is replicated on a distributed-memory ("shared-nothing" [35]) machine, it has to be sorted before replication or locally on each machine after replication. If it is replicated to processes on a shared-memory machine, it may be possible to scan the inner relation synchronously by all join processes, thus saving most of the I/O and some buffer memory (which can be used to increase the heap size) at the expense of increased synchronization needs.

Notice that on a shared-memory machine, it may be advantageous to use only one heap (or very few heaps) for the outer relation because the larger the heap is, the longer the runs will be and the fewer passes over the inner relation will be required. Thus, depending on the concurrent access protocol to this heap, it may not be possible to exploit all available parallelism on a highly parallel shared-memory machine. This caveat is entirely analogous with the detrimental effect of memory division in parallel external sorting on such machines [15].

4. Analytical Performance Comparison

In this section, we will compare merge join, heap-filter merge join, and hybrid hash join. As mentioned before, we assume that the inputs are unsorted. We omit the cost of reading the unsorted inputs and writing the output to disk since these costs are equal for all algorithms and do not assist in the comparison.

Figure 1 shows the join cost for the three algorithms discussed here for a memory size of 100 pages and inner input size of 250 pages. The outer input size varies from 250 to 10,000 pages. All three algorithms have linear cost functions because we assumed a single level merge for all sort operations and single-level (not recursive) hash table overflow resolution. It is obvious that merge join is inferior to hybrid hash join. However, the difference between heap-filter merge join and hybrid hash is probably surprising for most readers. The reason, as pointed out above, is that no part of the larger, outer input is ever written to temporary disk files.

We have to admit that we carefully selected the inner-to-memory ratio for this graph. If the ratio is below two, the cost of hybrid hash join is less because only sequential I/O is necessary to write the overflow file. If the ratio is too high, the cost of repetitive scans becomes dominating.

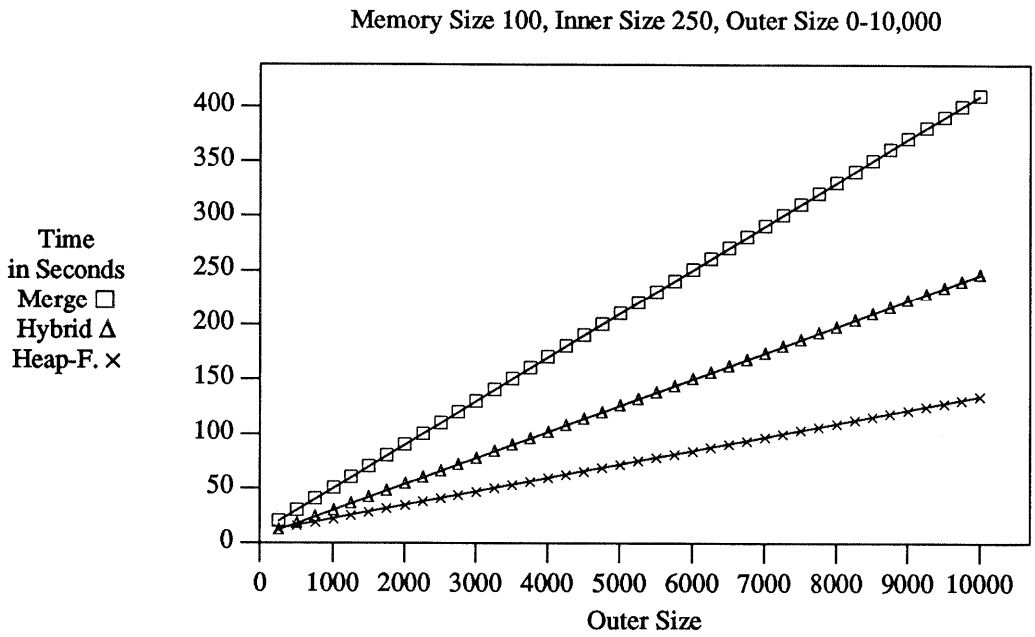


Figure 1. Join Costs depending on Outer Input Size.

In fact, if this ratio is above four, the number of I/Os required for merge join is less than that for heap-filter merge join. Considering the fact that we estimate triple the cost of sequential I/O for random I/O, the break-even point between these two algorithms is characterized by an inner input size eight times the size of memory.

To get a more realistic view of heap-filter merge join, we fixed the outer input and memory sizes and varied the inner input size from 100 to 1,000 pages. Figure 2 shows the join costs for the three algorithms depending on the inner input size. Merge join is most expensive over most of the range shown, dominated by the sort cost for the large outer input. The break-even point between merge join and heap-filter merge join is at 800 pages for the inner input, eight times the memory size.

The cost curve for hybrid hash join is the most interesting: This algorithm is superior if there is little overflow, but the cost is substantial if the large outer input must be partitioned into multiple overflow files requiring random I/O. Only when the inner input size is a large multiple of the memory size, hybrid hash join becomes superior to heap-filter merge join. Clearly, the

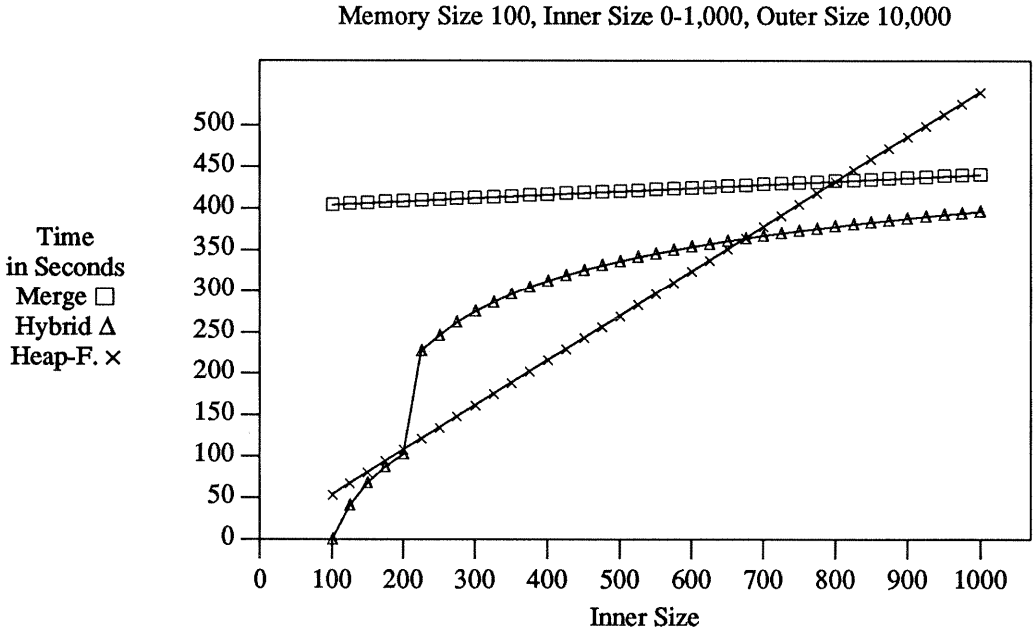


Figure 2. Join Costs depending on Inner Input Size.

asymptotic cost of hybrid hash join is superior to both of the other algorithms, but there seems to be a substantial window in which heap-filter merge join outperforms hybrid hash.

To illustrate this window, we fixed the memory size and varied both inner and outer input sizes. In Figure 3, we shaded the area in which heap-filter merge join outperforms hybrid hash join. As can be seen from Figure 3, this area is substantial for medium-size inputs, i.e., where the inner input size is a small multiple of the outer input size. Basically, for medium-size inner relations (about two to six times as large as memory, 220 to 660 pages), heap-filter merge join dominates hybrid hash join for all but the smallest outer inputs (< 500 pages). For medium-size outer inputs (500 to 2,500 pages) and not-so-small inner inputs (480 to 650 pages) hybrid hash join outperforms heap-filter merge join because the I/O cost for repetitive scans is higher than the cost of writing parts of the outer input to overflow files.

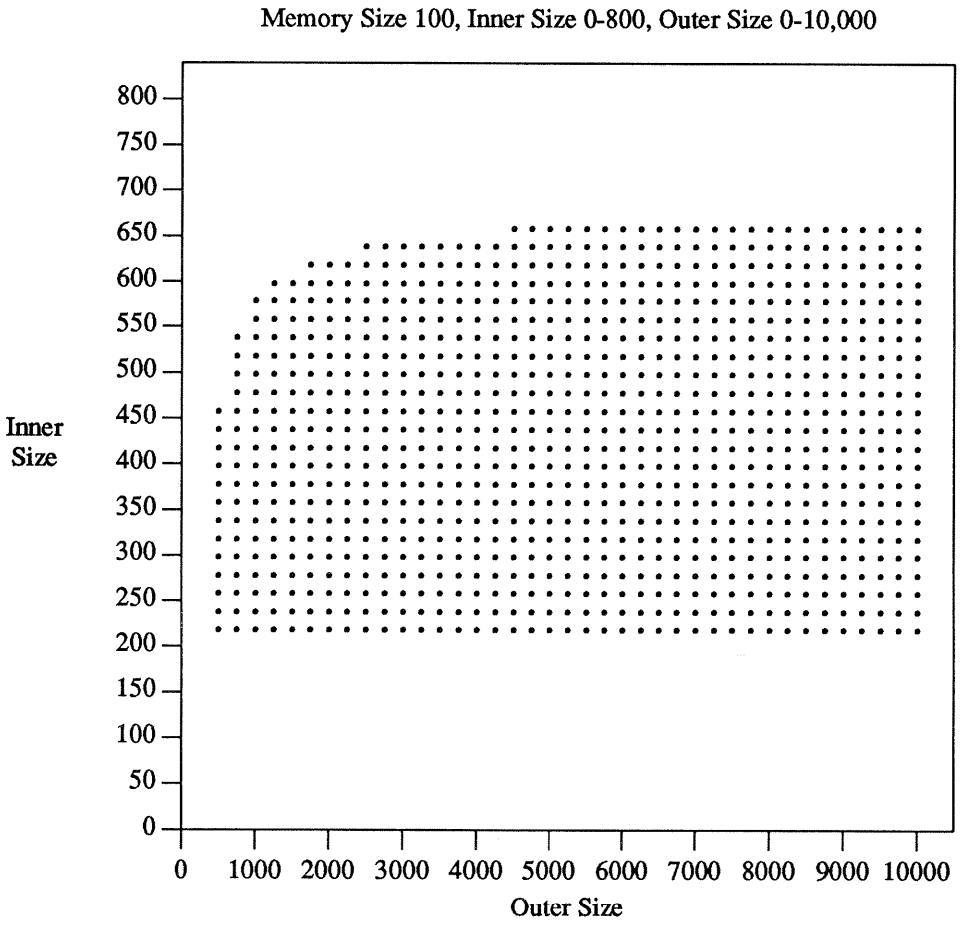


Figure 3. Region where Heap-Filter Merge dominates Hybrid Hash.

5. Summary and Conclusions

We have described a new equi-join algorithm based on the well-known merge join algorithm, which we call *heap-filter merge-join.* For moderately large inputs, it outperforms merge join by a significant margin. When compared to hybrid hash join, commonly regarded as a very efficient algorithm, heap-filter merge join is superior in some parameter ranges, namely if the inner input size is a small multiple of the memory size.

As hybrid hash join, heap-filter merge join depends on randomness of data to perform well. In hybrid hash join, if the hash values are not uniformly distributed, the partitions may not be of even size and require recursive overflow resolution which may lead, in the absolutely worst case,

to the rather inefficient *simple hash join* [6, 31]. In heap-filter merge join, if the keys in the outer are not random but somewhat presorted in reverse order, the number of outer runs may be larger than estimated in the cost comparisons. However, while hybrid hash join may deteriorate by a factor equal to the number of pages in memory[4], heap-filter merge join will deteriorate at most by a factor of two since the runs will be at least as large as memory. Furthermore, if the outer keys are somewhat presorted in correct order, heap-filter merge join will produce runs larger than estimated above and the I/O cost due to repetitive passes over the inner input can be significantly reduced. In the best case, only a single run is produced and heap-filter merge join becomes true merge join with only a single scan of the sorted inner input.

The results of this study have surprised us; we expected heap-filter merge join to be inferior to hybrid hash join for all input sizes, or to be marginally superior in a very narrow range, but we found that this range is actually not very narrow and overlaps with the range in which the difference of hash and merge join algorithms is most pronounced; thus, the new algorithm may help understanding and closing the gap between sort- and hash-based query processing algorithms.

References

1. C. K. Baru, O. Frieder, D. Kandlur and M. Segal, "Join on a Cube: Analysis, Simulation, and Implementation", *Proceedings of the 5th International Workshop on Database Machines*, 1987.
2. M. Blasgen and K. Eswaran, "On the Evaluation of Queries in a Relational Database System", *IBM Research Report*, San Jose, CA., April 8, 1976.
3. M. Blasgen and K. Eswaran, "Storage and Access in Relational Databases", *IBM Systems Journal 16*, 4 (1977).
4. K. Bratbergsengen, "Hashing Methods and Relational Algebra Operations", *Proceedings of the Conference on Very Large Data Bases*, Singapore, August 1984, 323-333.
5. 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.
6. 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.
7. 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.

[4] Actually, the worst factor for non-recursive overflow resolution is determined by the maximum fan-out which is limited by the memory size in pages. This factor can be even worse for very large inputs that require recursive hash table overflow resolution, even if the partitioning function is perfect.

8. D. DeWitt and D. Schneider, "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.

9. B. C. Desai, "Performance of a Composite Attribute and Join Index", *IEEE Transactions on Software Engineering SE-15*, 2 (February 1989), 142.

10. R. Epstein, M. Stonebraker and E. Wong, "Distributed Query Processing in a Relational Database System", *Proceedings of the ACM SIGMOD Conference*, Austin, TX., May 1978.

11. R. Epstein and M. Stonebraker, "Analysis of Distributed Data Base Processing Strategies", *Proceedings of the Conference on Very Large Data Bases*, Montreal, Canada, October 1980, 92-101.

12. 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.

13. P. Goyal, H. F. Li, E. Regener and F. Sadri, "Scheduling Page Fetches in Join Operations Using Bc-Trees", *Proceedings of the IEEE Conference on Data Engineering*, Los Angeles, CA., February 1988, 304-310.

14. G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System", *Proceedings of the ACM SIGMOD Conference*, Atlantic City, NJ., May 1990.

15. G. Graefe, "Parallel External Sorting in Volcano", *submitted for publication*, February 1990.

16. W. Kim, "A New Way to Compute the Product and Join of Relations", *Proceedings of the ACM SIGMOD Conference*, Santa Monica, CA., May 1980, 179-187.

17. M. Kitsuregawa, L. Harada and M. Takagi, "Join Strategies on KD-Tree Indexed Relations", *Proceedings of the IEEE Conference on Data Engineering*, Los Angelos, CA, February 1989, 85.

18. D. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison-Wesley, Reading, MA., 1973.

19. R. P. Kooi, "The Optimization of Queries in Relational Databases", *Ph.D. Thesis*, September 1980.

20. H. Lu and M. Carey, "Some Experimental Results on Distributed Join Algorithms in a Local Network", *Proceedings of the Conference on Very Large Data Bases*, Stockholm, Sweden, August 1985, 292-304.

21. M. J. Menon and D. K. Hsiao, "Design and Analysis of a Relational Join Operation for VLSI", *Proceeding of the Conference on Very Large Data Bases*, Cannes, France, September 1981, 44-55.

22. T. H. Merrett, Y. Kambayashi and H. Yasuura, "Scheduling of Page-Fetches in Join Operations", *Proceeding of the Conference on Very Large Data Bases*, Cannes, France, September 1981, 488-498.

23. K. P. Mikkilineni and S. Y. W. Su, "An Evaluation of Relational Join Algorithms in a Pipelined Query Processing Environment", *IEEE Transactions on Software Engineering SE-14*, 6 (June 1988), 838.

24. M. C. Murphy and D. Rotem, "Effective Resource Utilization for Multiprocessor Join Execution", *LBL Techical Report 26601* (1989).

25. T. Nakayama, M. Hirakawa and T. Ichikawa, "Architecture and Algorithm for Parallel Execution of a Join Operation", *Proceedings of the IEEE Conference on Data Engineering*, Los Angeles, CA., April 1984, 160.

26. M. Nakayama, M. Kitsuregawa and M. Takagi, "Hash-Partitioned Join Method Using Dynamic Destaging Strategy", *Proceedings of the Conference on Very Large Databases*, Los Angeles, CA, August 1988, 468-478.

27. G. Z. Qadah and K. B. Irani, "The Join Algorithms on a Shared-Memory Multiprocessor Database Machine", *IEEE Transactions on Software Engineering SE-14*, 11 (November 1988),

1668.

28. G. Z. Qadah, "Filter-Based Join Algorithms on Uniprocessor and Distributed-Memory Multiprocessor Database Machines", *Lecture Notes in Computer Science 303* (April 1988), 388, Springer Verlag.

29. J. P. Richardson, H. Lu and K. Mikkilineni, "Design and Evaluation of Parallel Pipelined Join Algorithms", *Proceedings of the ACM SIGMOD Conference*, San Francisco, CA., May 1987, 399-409.

30. B. Salzberg, "Merging Sorted Runs Using Large Main Memory", *Acta Informatica 27* (1990), 195-215, Springer International.

31. L. D. Shapiro, "Join Processing in Database Systems with Large Main Memories", *ACM Transactions on Database Systems 11*, 3 (September 1986), 239-264.

32. R. Shultz and I. Miller, "Tree Structured Multiple Processor Join Methods", *Proceedings of the IEEE Conference on Data Engineering*, Los Angeles, CA., February 1987, 190.

33. J. M. Smith and P. Y. T. Chang, "Optimizing the Performance of a Relational Algebra Database Interface", *Communications of the ACM 18*, 10 (October 1975), 568-579.

34. J. W. Stamos and H. C. Young, "A Symmetric Fragment and Replicate Algorithm for Distributed Joins", *Technical Report RJ7188* (December 5, 1989), IBM Almaden Research Lab. Instead of partitioning one join input and replicating the other, partition both and join each fragment from R with each fragment of S. E.g., on six processors, partition R in two and S in three fragments. This algorithm supports any theta-join. Specialized join methods will remain valid in special cases..

35. M. Stonebraker, "The Case for Shared-Nothing", *IEEE Database Engineering 9*, 1 (1986).

36. J. A. Thom, K. Ramamohanarao and L. Naish, "A Superjoin Algorithm for Deductive Databases", *Proceeding of the Conference on Very Large Data Bases*, Kyoto, Japan, August 1986, 189-196.

37. S. Todd, "PRTV: An efficient implementation for large relational data bases", *Proceedings of the Conference on Very Large Data Bases*, 1975, 554-556.

38. S. J. P. Todd, "The Peterlee Relational Test Vehicle – A System Overview", *IBM Systems Journal 15*, 4 (1976), 285-308.

39. P. Valduriez and G. Gardarin, "Join and Semijoin Algorithms for a Multiprocessor Database Machine", *ACM Transactions on Database Systems 9*, 1 (March 1984), 133-161.

40. P. Valduriez, "Join Indices", *ACM Transaction on Database Systems 12*, 2 (June 1987), 218-246.