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

Goetz Graefe

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 89-012

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

Goetz Graefe

Oregon Graduate Center
Beaverton, Oregon 97006-1999
graefe@cse.ogc.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 relations. In many cases, it even compares favorably with hybrid hash join.

1. Introduction

While there seems to be an abundance of relational join algorithms, some cases still allow improvements. In this technical note, 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 relation and merges them directly with the inner relation. This method avoids an substantial amount of I/O for sorting, at the expense of increased I/O for the inner relation.

In the following section, we discuss known algorithms, in particular 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, two variants of heap-filter merge join, and hybrid hash 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 [1].

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], and is the algorithm of choice for large relations in almost all commercial database systems. The idea is quite simple and well-known: Sort both relations on the join attribute[1], and merge them advancing a scan pointer in each relation. If both relations contain duplicate join attribute values, the scan pointer in the inner relation sometimes has to be backed up. We leave the exact scan logic to the reader as exercise or to look it up in a database text.

The major cost is for the sort step. For simplicity, we only concern ourselves with I/O costs, even though we realize that the CPU cost is 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 relation sizes; therefore we calculate the cost for only one merge level. For sorting the inner relation, using the cost parameters in Table 1 the cost is

$$sorti := (seq + rnd)\ inner.$$

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

inner	size of inner relation in pages
outer	size of outer relation 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.

$$(seq + rnd)\ outer + sorti$$

We will come back to this formula in Section 4.

2.2. Hybrid Hash Join

Hashing is a very fast method to find equality matches and a number of hash-based join algorithms have been proposed, e.g., [3]. 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 [4, 5, 6]. In a recent comparison of several algorithms, hybrid hash join was found to provide superior performance over a wide range of parameters [1].

Hybrid hash join is an optimistic hash join algorithm; we call it optimistic since it starts out with the assumption that the hash table overflow will fit into memory. When hash table overflow will occurs, a portion 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, another portion is dumped, etc. Thus multiple overflow files can be created and added to while building the hash table. Notice that if multiple overflows are dealt with more efficiently if multiple overflow files are used.

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 use the formula

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

Writing to overflow files is sequential if there is only one such file, otherwise it uses 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 input relations. Our effort was inspired by the desire to reduce the sorting costs. We assume in the sequel that both relations are originally unsorted.

We assume that the outer relation is the larger of the two relations. Almost all algorithms perform very well if the inner relation fits in main memory; therefore we will not concern ourselves with this case. Let us assume that the inner relation's size is a moderate multiple of the memory size, say twice to ten times the size of memory, and that the outer relation is quite large.

The new algorithm, which we call *heap-filter merge join*, avoids sorting the outer relation completely (which is different than completely avoiding the sort!) and instead joins the initial sorted runs immediately with the inner relation. 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 relation 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 relation repeatedly. If a heap is used for run generation, i.e., if all tuples from the outer relations 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 relation divided by twice the memory size [7]. The inner relation must be joined with each of these runs. Using the assumption that the inner relation is larger than memory, the inner relation must be retrieved repeatedly from disk, once for each run of the outer relation.

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 relation, at cost *sorti* developed above. Second, we need to scan the sorted inner relation once for each run from the outer relation. The number of these runs is equal to

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

The cost for each scan over the inner relation is

$$seq \ inner$$

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

$$R \ inner \ seq + sorti$$

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 relation for the next scan. Fortunately, this can easily be done by creating *alternating runs* from the outer relation. This means that the first run is ascending, the next one descending, the third one ascending again, etc. For these runs, the sorted inner relation can be scanned forward, backward, forward, backward, etc., using the last page of one scan as the first of the next, without I/O. Careful analysis will show that only one page should be used; using more memory pages will decrease the size of runs from the outer relation but increase the number of scans over the inner. For this *alternating heap-filter merge join*, the I/O cost is

$$(R\ (inner - 1) + 1)\ seq + sorti$$

3.2. Complex Queries

We would like to point out that the heap-filter merge join algorithms not only have good performance (as we will see in the next section), they also allow dataflow between operators in a complex query. For example, as soon as the first tuple from the outer input has traveled through the heap, it can be joined with the inner relation and an output tuple produced. Note that all algorithms discussed here consume one relation completely before starting to consume the other, and therefore before producing results.

4. Analytical Performance Comparison

In this section, we will compare merge join, heap-filter merge join, alternating heap-filter merge join, and hybrid hash join. As mentioned before, we assume that both input relations originally are not sorted. We omit the cost of reading the unsorted relations and writing the output to disk since these costs are equal for all algorithms. We only show the cost for the alternating variant of heap-filter merge join. The difference between heap-filter merge join and alternating heap-filter merge join is minimal since the crucial factor in the formula is 249 vs. 250, respectively.

Figure 1 shows the join cost for the four algorithms discussed here for a memory size of 100 pages and inner relation size of 250 pages. The outer relation sizes varies from 0 to 10,000 pages. All four algorithms have linear cost functions because we assumed a single level merge for all sort operations. 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 relation is ever written to temporary disk files.

We have to admit that we selected the inner-to-memory ratio carefully for this graph. If the ratio is below two, the cost of hybrid hash join is much 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

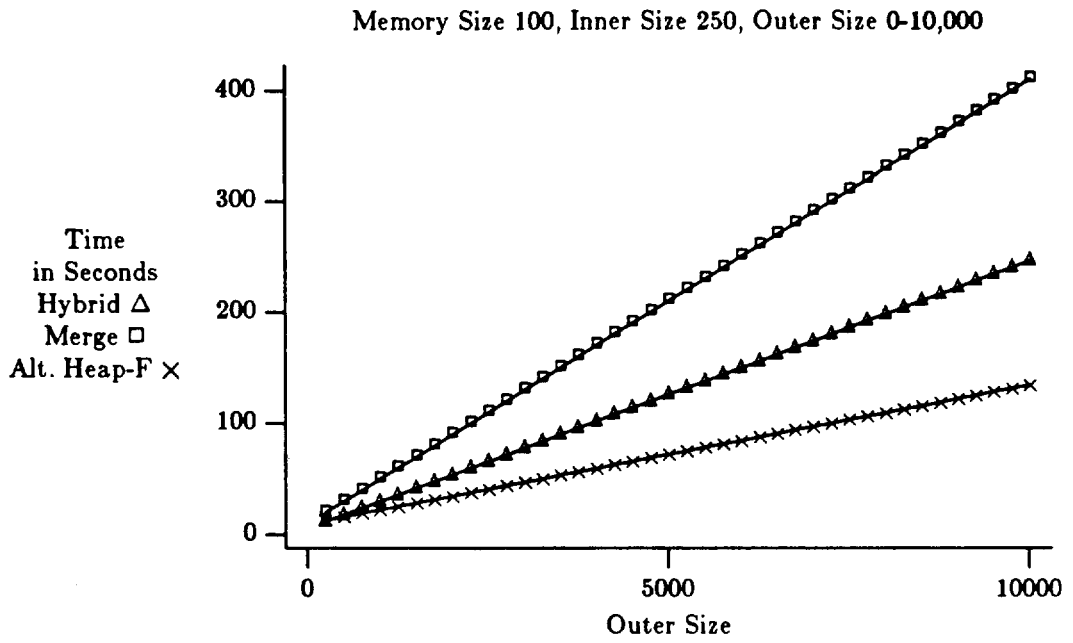

Figure 1. Join Costs depending on Outer Relation Size.

dominating. In fact, if this ratio is above four, the number of I/Os for merge join is less than for heap-filter merge join. Considering 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 relations eight times the size of memory.

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

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

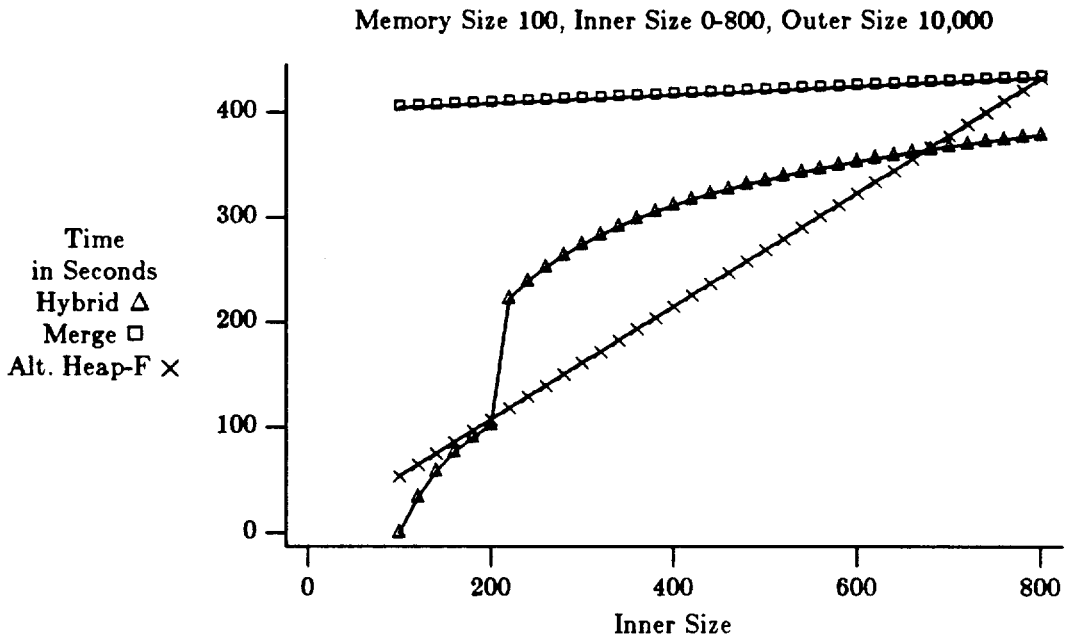

Figure 2. Join Cost depending on Inner Relation Size.

asymptotic cost of hybrid hash join is superior to any 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 relation sizes. In Figure 3, we shaded the area in which alternating heap-filter merge join outperforms hybrid hash join. As can be seen from the figure, this area is substantial for medium-size relations, i.e., where the inner relation size is a small multiple of the outer relation size.

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 relations, 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 relation size is a small multiple of the memory size.

The results of this study have surprised us; we expected to heap-filter merge join to be inferior for all relation sizes, to be marginally superior is a very narrow range. While we believe

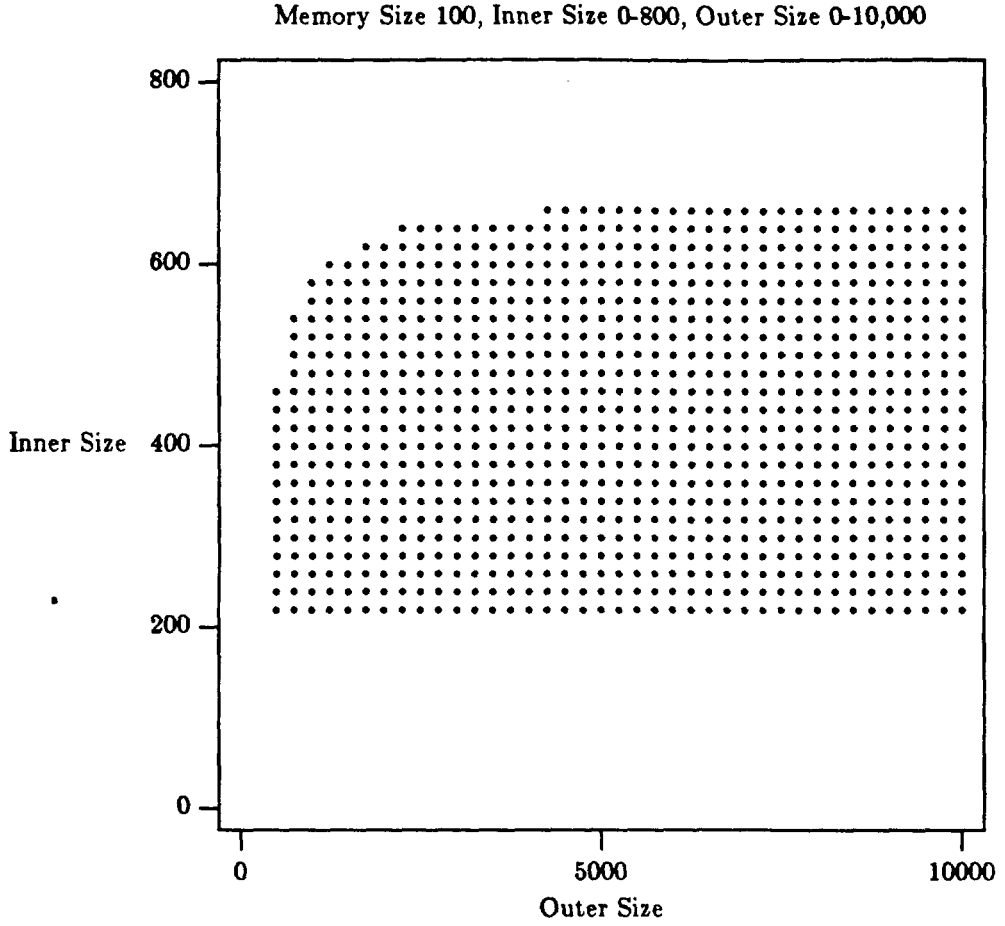

Figure 3. Region where Alternating Heap-Filter dominates Hybrid

we used reasonably realistic cost functions, we intend to verify this study by implementing and comparing these algorithms in the framework of the Volcano query processing system [8].

References

1. 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*, p. 110 (May-June 1989).
2. M. Blasgen and K. Eswaran, "Storage and Access in Relational Databases," *IBM Systems Journal* **16**(4)(1977).
3. K. Bratbergsengen, "Hashing Methods and Relational Algebra Operations," *Proceedings of the Conference on Very Large Data Bases*, pp. 323-333 (August 1984).
4. 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*, pp. 1-8 (June 1984).
5. L.D. Shapiro, "Join Processing in Database Systems with Large Main Memories," *ACM Transactions on Database Systems* **11**(3) pp. 239-264 (September 1986).

6. 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*, pp. 209-219 (August 1986).

7. D. Knuth, *The Art of Computer Programming*, Addison-Wesley, Reading, MA. (1973).

8. G. Graefe, "Volcano: An Extensible and Parallel Dataflow Query Processing System," *Oregon Graduate Center, Computer Science Technical Report*, (89-006)(June 1989).