Cache Conscious Algorithms for Relational Query Processing

Ambuj Shatdal
Chander Kant
Jeffrey F. Naughton

Technical Report #1234

June 1994

Cache Conscious Algorithms for Relational Query Processing*

Ambuj Shatdal Chander Kant Jeffrey F. Naughton
Computer Sciences Department
University of Wisconsin-Madison
{shatdal,ck,naughton}@cs.wisc.edu

Computer Sciences Technical Report # 1234

June, 1994

Abstract

The current main memory (DRAM) access speeds lag far behind CPU speeds. Cache memory, made of static RAM, is being used in today's architectures to bridge this gap. It provides access latencies of 2–4 processor cycles, in contrast to main memory which requires 15–25 cycles. Therefore, the performance of the CPU depends upon how well the cache can be utilized. We show that there are significant benefits in redesigning our traditional query processing algorithms so that they can make better use of the cache. The new algorithms run 8%–200% faster than the traditional ones.

1 Introduction

The DRAM access speeds have not reduced much compared to the CPU cycle time reduction resulting from the improvements in VLSI technology. Cache memories, made of fast static RAM, help alleviate this disparity by exploiting the spatial and temporal locality in the data accesses of a program. However, programs with poor access locality waste significantly many cycles transferring the data to and from the cache memory resulting in poor CPU performance.

The above observation makes it important that the algorithms for various relational operations should be designed to take maximum advantage of the cache memory. In this paper we study the existing algorithms in terms of their cache performance. We redesign the traditional algorithms by taking into account the fact that modern computers have CPU caches. However, this paper is not about query optimization incorporating the cache memory as a parameter. In this paper we concentrate on the join and aggregation algorithms.

The conventional wisdom in database community is that query evaluation touches so much data that locality in data accesses is inherently poor. Further, there is a widespread but false belief that once data is in memory it is accessed as fast as it could be. We have challenged the conventional beliefs by showing that by designing cache conscious algorithms one can significantly speed up the CPU processing portion of query processing. For main-memory database systems (or largely-memory resident database systems) this is very significant. Further, the recent work by Nyberg et al. [NBC+94] suggests that the I/O response time can be reduced through the use of software assisted disk striping thus making the CPU cost of the query processing dominate i.e. a relation can be read into the memory faster than it is processed. This clearly makes cache optimizations, which speed up CPU processing, extremely relevant for disk-resident data also.

In related work, Nyberg et al. [NBC+94] have shown that for achieving high performance sorting, one should worry about cache memory. They have emphasized a large cache and do not explore alternative optimization techniques. In some sense, our work picks up where they have left off. We show how we can incorporate cache memory in the design process of the algorithm and not as an afterthought. We do not argue for very large caches but show that given any size cache, our techniques are useful. This paper does not propose that a completely different algorithm be designed for each hardware platform. Rather the proposal is that the same algorithm can be ported on different platforms after a phase of performance tuning using some cache profiler.

Once the designer is aware of the presence of cache and its behavior, some techniques do not seem arcane. However, in general, cache behavior is fairly complex and one needs to use some cache profiling tool like *cprof* [LW94] as an aid to study the cache behavior of a particular algorithm. Sometimes, we find ourselves revisiting some of the same optimizations in a different guise as made for the memory/disk portion of the memory hierarchy. At the same time one must note that these two are not equivalent problems. Among the major differences are: cache is entirely hardware managed and user has no direct control over what resides in

*The paper also appears in the Proceedings of the 20th International Conference on Very Large Databases, Santiago, Chile, September 1994. This research was supported by NSF grant IRI-9157357.

cacne; caches are not fully associative unlike disk cache in main memory; and lastly we can not, in general, trade off CPU cycles for improving cache performance which is the most important difference between memory/disk and cache/memory optimizations.

The rest of the paper is organized as follows. Section 2 briefly reviews cache memories. Section 3 describes some known techniques for cache optimizations. In section 4 we present the query processing algorithms and study how cache optimization helps. Section 5 offers our conclusions.

2 Overview of Cache Memories

Cache memories are small, fast static RAM memories that improve program performance by holding recently referenced data [Smi82]. Memory references satisfied by the cache, called hits, proceed at processor speed; those unsatisfied, called misses, incur a cache miss penalty and have to fetch the corresponding cache block from the main memory. The management of the cache is done entirely by the hardware with no direct user control.

Unlike other levels of the memory hierarchy, caches are sometimes divided into instruction-only and data-only caches. Separate caches offer the opportunity of optimizing each cache separately. In this study we constrain ourselves to data cache performance.

Caches are characterized by three major parameters: Capacity (C), Block Size(B) and Associativity (A).

Capacity A cache's capacity (C) simply defines the total number of bytes it may contain.

BlockSize The block size (B) determines how many contiguous bytes are fetched on each cache miss. A cache block exploits spatial locality by (pre-)fetching multiple contiguous words (thus reducing chances of a future miss), a cache block, whenever a miss occurs.

Associativity Associativity refers to the numbers of unique places in the cache a particular block may reside in. If a block can reside in any place in the cache (A=C/B) we call it a fully-associative cache, if it can reside in exactly one place (A=1) we call it direct mapped, if it can reside in exactly A places, we call it A-way set associative. In associative caches, LRU replacement policy is used to decide which cache block will be replaced. Most caches, in practice, are either direct mapped or have very small set-associativity.

Cache misses can be categorized into following three disjoint types [HS89]. The relation of the cache miss types to the cache characteristics is also described.

Compulsory A reference that misses because it is the very first reference to a cache block is classified as a compulsory miss. By definition, compulsory misses can not be reduced without changing the basic algorithm. However, larger cache block size will decrease the number of compulsory misses as more data will be prefetched in a sequential access pattern.

Capacity A reference that misses in a fully associative cache is classified as a capacity miss because the finite sized cache is unable to hold all the referenced data. Capacity misses can be minimized by increasing temporal and spatial locality of references in the algorithm. Increasing cache size also reduces the capacity misses because it captures more locality.

Conflict A reference that hits in a fully associative cache but misses in an A-way set associative cache is classified as a conflict miss. This is because even though the cache was large enough to hold all the recently referenced data, its associativity constraints forced some of the required data out of the cache prematurely. Conflict misses are the hardest to remove because they occur because of address conflicts in the data structure layout and are specific to a cache size and associativity. Data structures would, in general, have to be remapped so as to minimize conflicting addresses. Increasing the associativity of a cache will decrease the conflict misses.

A cache profiler like *cprof* [LW94] finds the cache behavior of an algorithm by simulating all data accesses in the appropriately configured cache. The address traces are generated by an instruction level profiler like *qpt* [Lar93] which are fed to a cache simulator which finds the requisite properties of the data references. *Cprof* classifies the cache misses in the above categories by every line of code and by every data structure. The programmer can view the cache profile and find the cache behavior of the program in detail. However, since this simulation is based on virtual addresses, the results are only approximately true for a physically addressed cache (like the level 2 cache in the DEC 3000) as non-conflicting virtual addresses may conflict in a physically mapped cache. [KH92] shows that this effect is minor in most cases (especially when virtual address space is much larger than the cache size, which holds for typical database applications) and operating systems can use simple techniques to overcome the problem of introduction of cache conflicts due to virtual to physical memory mapping.

3 Optimization Techniques

The main aim of optimizing algorithms for cache memory is to ensure that as few cache misses occur as possible without significantly increasing the number of instructions executed i.e. the CPU overhead. In this study, however, we do not concern ourselves with optimizations which depend on the exact cache configuration in terms of block size and associativity i.e. we do not attempt to answer the question how algorithms can exploit the associativity or block size per se. The reason is that changing block size only affects compulsory misses, which can not be removed given a particular cache. Associativity does help remove conflict misses. However, we find that conflict misses are relatively few compared to capacity misses and hence not very significant in terms of performance.

Many opportunities for cache optimization are not at all obvious at the outset. Many times it is not possible to find the cache bottleneck in a code by just looking at it and by looking at the CPU runtime. A cache profiler is usually needed to find the possibilities for improvement because it localizes the optimization space, i.e. one could concentrate on thinking of the optimizations so as to remove the cache misses occuring at that point in the program. As mentioned earlier, we used *cprof* for our studies. In the following sections we study some specific techniques for removing the cache misses.

3.1 Blocking

In blocking, an algorithm is restructured to reuse chunks of data that fit in the cache. Take the example of naive nested loops for computing a non-equijoin. In case of disk resident databases the reason for switching to nested loops with blocking is to significantly reduce disk I/O. However, the motivation to block nested loop in case of a memory resident relation does not seem apparent at first because of the (incorrect) belief that memory accesses are of uniform speed. We found that blocking indeed improves performance when tuples are blocked such that a block of tuples fits entirely in the cache. This is expected because now accesses to the tuples of inner relation suffer significantly fewer cache misses as we process a block entirely before discarding it.

Example 3.1

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    process(a[i],b[j])
```

when blocked on array b, will look like

```
for (bkNo = 0; bkNo < N / BKSZ; bkNo++)
  for (i = 0; i < M; i++)
    for (j = bkNo*BKSZ; j < (bkNo+1)*BKSZ; j++)
      process(a[i],b[j])
```

which will ensure that the elements of array b in a block will almost always be in the cache provided BKSZ is less than the cache size, thus significantly improving the cache performance. □

3.2 Partitioning

Another technique is to distribute the data in partitions as in external sorting. These partitions are created such that each partition fits in the cache. AlphaSort [NBC$^+$94] uses this technique to speedup the in-memory sorting. There is an overhead of creating the partitions but in most cases the benefit gained overshadows it. In many database algorithms, a good way of generating partitions is by hash partitioning the relations. For example, in joins, hash partitioned partitions can be joined independently. Consider the simple example of sorting:

Example 3.2

```
quicksort(relation[N])
```

is changed to

```
partition relation into blocks < cache size
for each partition r
  quicksort(relation[PARTITIONSIZE]);
merge all the partitions
```

Now, since the entire partition of the array being sorted fits in the cache, the quicksort runs significantly faster as there are few cache misses. This more than compensates for the extra merge step resulting in greater overall sorting speed (for large enough N). This is the essence of the in-memory AlphaSort [NBC$^+$94] algorithm. □

Note that blocking and partitioning are distinct techniques. In blocking we restructure the algorithm and do not change the layout of the data, whereas in partitioning we reorganize the layout of the data to make maximum use of the cache.

3.3 Extracting Relevant Data

Reducing the data which needs to be accessed is another effective technique. In sorting, for example, instead of sorting whole records one could extract the sorting key and a pointer to record and use that for sorting. Smaller size data effectively means that more relevant data can fit in the cache. One must note that extracting relevant data can also be a memory optimization, in the sense that less memory has to be accessed (and that it would be faster, though not as much, even in absence of a cache). This optimization can be taken a little further as in using only key prefixes instead of keys for sorting [NBC$^+$94].

For example, from a 200 byte record, we could extract an 12 byte key and a 4 byte pointer to do the sorting.

Sorting with smaller records will be significantly faster as the cache will not get irrelevant data.

3.4 Loop Fusion

Many a time separate loops operating on a data structure can be merged resulting in better locality of reference for the data structure. The example below combines creation of a key pointer array (see above section) and building a hash table.

Example 3.3

```
for (i = 0; i < N; i++)
  a[i].key = relation[i].key;
  a[i].ptr = relation[i].ptr;
for (i = 0; i < N; i++)
  insert_in_hashtable(a[i]);
```

is changed to

```
for (i = 0; i < N; i++)
  a[i].key = relation[i].key;
  a[i].ptr = relation[i].ptr;
  insert_in_hashtable(a[i]);
```

which will improve the likelihood that a[i] is in the cache when it is inserted in the hash table. □

3.5 Data Clustering

At the physical database design level, one can cluster the fields of a tuple in such a way that fields accessed contemporaneously are stored together. This results in better spatial locality when the two related fields of the tuple are accessed. For example, if in a relation a particular *group by* attribute always goes with another attribute on which the aggregation is performed, then both should be allocated next to each other so that the access to *group by* attribute may prefetch the aggregation attribute of the tuple.

In this study we have concentrated on reducing capacity misses. Hence our efforts are more focused on improving the temporal and spatial locality of the memory accesses rather than coming up with an optimal memory layout of relations.

4 Performance Evaluation

In this section we study: 1. how do we cache optimize a query processing algorithm and 2. what difference do these optimizations make.

We ran our performance tests on four different machines which are representative of the modern trends in microprocessor technology. These are the DECstation 5000/125, the DEC 3000/300, the HP Apollo 9000/710, and the SUN Sparcstation 10/51. We used the native compilers (except the SUN on which we used the gcc) with maximum practical optimization levels. Table 1 details the configuration of the machines we used. Here we must observe that cache access characteristics depend significantly on the compiler and optimization level used.

While we assume that the data operated on by the algorithms is memory-resident, we wanted to approximate the data layout in memory that would result in reading a page of tuples into a buffer pool. Accordingly, we stored the tuples in memory in slotted pages; the overhead of processing tuples in slotted pages (rather than packed arrays of tuples) is included in all of our results. Under these assumptions, in practice and in our study, an algorithm starts its processing on the relations stored in the buffer pool. The join result is left in the form of an in-memory join index [Val87]. In section 4.4 we discuss the tradeoffs and options in the generation of result relation. Furthermore, we incorporated the optimization of extracting the join attribute (*group by* attribute in case of aggregation) from the tuples for the processing whenever it was appropriate.

For our join algorithms both the relations had 50000 tuples each. Each tuple is 100 bytes long. Relations were generated in such a manner that each tuple in the first relation joined with approximately one tuple in the second. For aggregate processing, the relation had 100000 tuples of 100 byte length. The number of groups was 20. In section 4.3 we show that our results hold even when we vary these parameters. All reported timings are in seconds.

4.1 Case Study: Optimizing Hash Joins

We describe how we optimized the basic in-memory hash join algorithm [DKO+84] in detail on the DECstation 5000/125. We used the cache profiler *cprof* [LW94] to gain detailed information about the cache performance of the algorithms. That guided us quickly to the code having poor cache behavior and thus exposed the opportunities for optimization.

Assume R and S are the two relations (or fragments of relations of a bigger join) being joined which are now in the main memory in slotted pages after having been read from the disk.

The in-memory hash join algorithm works as follows. First a hash table of the tuples of R is created by hashing on the join attribute. Then the tuples of the relation S are probed by hashing them on the join attribute and searching the attribute value in the hash table. Thus the implementation of the basic hash join algorithm looks like.

Algorithm 4.1 *BaseHash(R,S)*

```
BuildHashTable(H[R]);
for each s in S
  Probe(s, H[R]);
```

Machine	Microprocessor	Data Cache Size
DECstation 5000/125	MIPS R3000	64K
SUN Sparcstation 10/51	SPARC/Viking	16K (on-chip) + 1M
DEC 3000/300	Alpha AXP A	8K (on-chip) + 256K
HP Apollo 9000 Series 700/710	HPPA-RISC 1.1	64K

Table 1: Machine Configurations

Upon profiling, the algorithm showed significant number of cache misses. Table 2 shows the cache misses suffered by each step of the algorithm. Building the hash table directly on the relation tuples suffers many cache misses because the entire hash table is unable to fit in the cache. Also, useless data is brought into the cache because the cache block prefetching brings in attributes not required for the join computation resulting in wastage of cache capacity. The probe phase has even more cache misses because every probe generates a random address in the hash table which is unlikely to be in the cache, coupled with the fact that accessing the probing tuple itself might result in a cache miss. This is only worsened by the cache pollution due to prefetching of irrelevant data since we are building the hash table from base tuples.

Step	comp.	capacity	conflict	Total
Build	37500	118731	2181	158412
Probe	25159	193137	2352	220648
Overall	62659	311868	4533	379060

Table 2: Cache Misses in BaseHash Join

First optimization which seems possible is to do attribute/pointer extraction in the building relation. Note that the building relation is accessed possibly several times, once for building and again in the probing phase, whereas the probing relation is accessed only once. Thus by doing the extraction, we increase the locality of accesses in both build and probe phases for the building relation. The locality improves because now there is no pollution of the cache which happens due to automatic prefetching of the spatially contiguous data as in the base case. This results in the following algorithm.

Algorithm 4.2 *Extraction(R,S)*

```
for each r in R
  ExtractKeyPointers(r)
  BuildHashTable(r)
Probe(s, H[R]);
```

As evident from the profile in table 3, this algorithm shows reduced number of cache misses in the build and the probe phase from the basic algorithms. The overhead of attribute/pointer extraction is more than compensated for the reduction in cache misses in the build and probe phases thus reducing the total number of cache misses. This reduction in cache misses results in a speedup of 7.2% over the basic hash join.

Step	comp.	capacity	conflict	Total
Extract(R)	25000	50000	5	75005
Build	25000	42692	1791	69483
Probe	25159	165491	6055	196705
Overall	75159	258183	7851	341193

Table 3: Cache Misses in Extraction Join

We still find that there are far too many cache misses in the building and probing phases. One strategy is to ensure that the built hash table is always kept in the cache thus reducing the cache misses in the building and probing phases. This can be implemented by dividing the relation into several (hash partitioned) partitions such that the hash tables built from these partitions would fit in the cache. This partitioning of the relation is done along with the attribute/pointer extraction (in both relations). Of course, this incurs the overhead of creating the partitions and processing them but the improvement due to reduction in cache misses more than compensates for that. Thus we come up with the following algorithm[1].

Algorithm 4.3 *PartitionedHash(R,S)*

```
ExtractKeyPointers_And_Partition(R)
ExtractKeyPointers_And_Partition(S)
for each partition i
  BuildHashTable(H[R[i]])
  for each s in S[i]
    Probe(s,H[R[i]])
```

Table 4 shows the cache misses suffered in this algorithm. We note that there are quite many cache misses in partitioning the relations. However, as expected, the joining of the partitions themselves suffer far fewer cache

[1] A keen observer would note that this is analogous to the GRACE algorithm [DKO+84] for join processing of disk resident relations.

misses as the building and probing partitions can be entirely cache resident. With this sharp a reduction in cache misses we would expect a relatively large speedup. However, as mentioned before, the extra processing involved in partition creation and processing reduces some of the advantage thus gained. The obtained speedup of the algorithm is 6.6%.

Step	comp.	capacity	conflict	Total
Partition(R)	37514	51250	174	88938
Partition(S)	25045	51212	244	76501
Build	2192	40445	9837	52474
Probe	25118	13860	27146	66124
Overall	89869	156767	37401	284037

Table 4: Cache Misses in PartitionedHash Join

Table 5 summarizes the results of the cache optimizations for hash join. Here (and elsewhere), speedup indicates the speedup over the basic algorithm.

Algorithm	Cache Misses	Time	Speedup
Base	379060	0.699	—
Extraction	341193	0.652	7.2%
Partitioned	284037	0.656	6.6%

Table 5: Optimizations for the Hash Join

Note that the number of compulsory cache misses actually increase with the optimizations. This implies that an infinite (or no cache) would actually show that the basic algorithm is the best. However, this also shows the importance of cache optimization, because it demonstrates that theoretically similar algorithms can have significantly differing performance depending on the way they utilize the cache. And lastly, it shows that one can not, in general, tradeoff CPU cycles for cache optimization as the advantage gained by a decrease in cache misses can be quickly nullified by the CPU overhead as evident by the PartitionedHash algorithm.

Finally, we measured the performance of these on the other machines to show that these techniques are not specific to any particular machine but hold in general. The timing speedup obtained on them are given in table 6.

Even though we report performance on different machines, when comparing the relative effect of cache optimizations on different architectures one must be careful not to ascribe all differences in the performance to properties of the processor, memory, and its cache. While these hardware parameters do affect the efficiency of the optimizations, we also found that factors such as the compiler used also have as strong an effect—we observed significant differences in the impact of the cache optimizations within a single machine by varying the compiler optimization level.

4.2 Other Query Processing Algorithms

We went through the above optimization process for other query processing algorithms, viz. the sort merge join, the nested loop join, the hash based aggregation and the sort based aggregation. In this section we describe the algorithms, point out the optimizations we made and the speedup obtained on the four machines.

4.2.1 The Sort Merge Join

The in-memory sort merge join [BE77] works as follows. First, both relations R and S are sorted on the join attribute by using an efficient sorting mechanism e.g. quicksort. Then the sorted relations are merged and the matching tuples are output. As mentioned earlier, we use the optimization proposed in [NBC+94] to extract the join attribute and a pointer to the tuple.

The basic algorithm sorts both the relations and merges them.

Algorithm 4.4 *BaseSort(R,S)*

```
ExtractKeyPointers(R)
ExtractKeyPointers(S)
Sort(R)
Sort(S)
Merge(R,S)
```

In this algorithm, the sorting suffers several cache misses because none of the attribute-pointers of R or S are in the cache. First simple optimization which we implemented was to do the sorting immediately after the attribute/pointer extraction resulting in the following algorithm.

Algorithm 4.5 *ImmediateSort(R,S)*

```
ExtractKeyPointers(R)
Sort(R)
ExtractKeyPointers(S)
Sort(S)
Merge(R,S)
```

However, since both R and S are bigger than the cache, the sorting itself suffers several cache misses. The optimization is to make (hash partitioned) partitions of sizes such that the one partition of both relations will fit in the cache. This significantly reduces the cache misses suffered in the sorting phase. In the final step, each partition is merged pairwise. This optimization is similar to the PartitionedHash join algorithm above where smaller partitions ensure fewer cache misses.

Algorithm 4.6 *PartitionedSort(R,S)*

	DECst'n 5K/125		DEC 3k/300		HP 9k/720		SUN 10/51	
Algorithm	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
Base	0.699	—	0.203	—	0.472	—	0.349	—
Extraction	0.652	7.2%	0.198	2.5%	0.434	8.7%	0.291	19.9%
Partitioned	0.656	6.6%	0.186	9.1%	0.432	9.3%	0.324	7.7%

Table 6: Speedups obtained on the other machines

```
ExtractKeyPointers_And_Partition(R)
for each partition i
  Sort(R[i])
ExtractKeyPointers_And_Partition(S)
for each partition i
  Sort(S[i])
for each partition i
  Merge(R[i],S[i])
```

After looking at the cache profile of the PartitionedSort we notice that the cache misses could be further reduced in the merge phase by fusing the sorting and merging of each of the partitions i.e. instead of first sorting all and then merging all the partitions, we sort and immediately merge the partitions. This loop fusion results in the following algorithm.

Algorithm 4.7 *ImprovedSort(R,S)*

```
ExtractKeyPointers_And_Partition(R)
ExtractKeyPointers_And_Partition(S)
for each partition i
  Sort(R[i])
  Sort(S[i])
  Merge(R[i],S[i])
```

Table 7 shows the running times and the speedup shown by the algorithms on the four machines.

We note that the partitioning helps much more in the case of the sort merge join compared to the hash join because the sorting operation is much more memory intensive and computationally expensive i.e. the reduction in the number of cache misses is much larger because of the partitioning and the relative overhead of making the partition is correspondingly much smaller.

4.2.2 Non-equijoin Algorithms: Nested Loops

The nested loop algorithm is the most common way of handling non-equijoins. The in-memory version of nested loop is straightforward and takes $O(|R| * |S|)$ time. In the traditional way of thinking about database algorithms, we feel that nothing much can be done to improve the performance of the nested loop join once the relations are in memory. But we had a lot in store for us.

The basic algorithm is as follows.

Algorithm 4.8 *BaseNestedLoop(R,S)*

```
ExtractKeyPointers(R)
ExtractKeyPointers(S)
for each tuple r in R
  for each tuple s in S
    if join(r,s) then
      produce result
```

Upon looking at the cache profile, we quickly realized that we were incurring far too many cache misses than were necessary. This is because the sequential access of the inner relation S has poor cache locality. Blocking on the inner relation such that each block fits in the cache improves the locality of access for the inner relation. This is because the entire block is in cache while it is processed and therefore it suffers very few cache misses. In BaseNestedLoop, every access to the tuples of inner relation will probably result in a cache miss as the sequential access would replace the tuples before they could be reused. The blocked nested loop algorithm is as follows.

Algorithm 4.9 *BlockedNestedLoop(R,S)*

```
ExtractKeyPointers(R)
ExtractKeyPointers(S)
for each block b of S
  for each tuple r in R
    for each tuple s in b
      if join(r,s) then
        produce result
```

Table 8 shows the running times and the speed shown by the algorithms. The reason that the optimization on SUN 10/51 doesn't show much improvement is because of the improved performance of the base case as all the key pointers fit in the 1MB secondary cache even in the base case.

4.2.3 Aggregation Algorithms

Aggregation with the *group by* clause involves more than a simple scan of the participating relation [Eps79]. We consider the two popular aggregation algorithms: the hash based aggregation and the sort based aggregation.

Hash Based Aggregation The in-memory hash aggregation on relation R works as follows. We accumulate the result in a hash table by hashing the tuples of

	DECst'n 5K/125		DEC 3k/300		HP 9k/720		SUN 10/51	
Algorithm	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
Base	1.789	—	0.504	—	0.794	—	0.672	—
Immediate	1.769	1.1%	0.495	1.8%	0.793	1.3%	0.667	0.7%
Partitioned	1.344	33.1%	0.336	50.0%	0.648	22.5%	0.532	26.3%
Improved	1.301	37.5%	0.327	54.1%	0.640	24.1%	0.523	28.5%

Table 7: Optimizations for the Sort Merge Join

	DECst'n 5K/125		DEC 3k/300		HP 9k/720		SUN 10/51	
Algorithm	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
NestedLoop	2244.05	—	490.11	—	569.10	—	413.51	—
Blocked	741.54	202.6%	205.11	138.9%	305.71	86.2%	348.16	18.8%

Table 8: Optimizations for Nested Loop Algorithm

the relation R on the *group by* attribute and computing the cumulative sum and count (analogous information must be kept for other aggregate functions like min). At first sight, it seems that since there is little computation required to do the aggregation, there is little room for improvement. But we were taken by a surprise.

The basic algorithm builds a hash table of the result relation by hashing tuples of the R relation and accumulates the sum (and count) for aggregation purposes for every group.

Algorithm 4.10 *BaseHash(R)*

```
for each tuple t in R
  Hash(t)
  Insert/update the hashtable entry for the group
```

It was natural to attempt the Extraction optimization for the hash aggregation as it had worked well in the hash join.

Algorithm 4.11 *Extraction(R)*

```
for each tuple t in R
  ExtractKeyPointer(t)
  Hash(t)
  Insert/update the hashtable entry for the group
```

However, we found that it does not improve the performance of the hash aggregation. The reason being that in aggregation, the hash table is accessed only once (which is a compulsory cache miss) and therefore key pointer extraction does not help. In the join, in contrast, the hash table is accessed twice: once for build and again for probe, the key pointer extraction reduces the cache misses in the second access thereby improving the performance. This shows that cache optimizations can be subtle and specific to a particular algorithm.

We still noticed many cache misses when the aggregation attribute was accessed for accumulating the sum (and count). We then clustered the two and found a significant reduction in cache misses and improvement in performance. Of course, this is not an algorithmic optimization but something which can be taken care of at the level of physical database design.

Table 9 shows the running times and the speedup shown by the algorithms.

Sort Based Aggregation In the traditional sorting based approach to compute *group by* aggregation, first the relation is sorted on the *group by* attribute thus collecting the tuples of the same group together. Then the sorted relation is scanned producing tuples per group. The basic algorithm is then as follows.

Algorithm 4.12 *BaseSort(R)*

```
ExtractKeyPointer(R)
Sort(R)
for each t in R
  Initialize/update the group entry for the group
```

Since sorting suffers quite many cache misses, we decided to use the "partitions" optimization for sorting so that it suffers few cache misses. Hash partitioning is not really practical here because the number of groups could be very small. Hence, we make random partitions, sort and do aggregation on each of the partitions. However, this requires an extra step of merging these partition aggregates to form the global aggregate. But we find that this extra overhead is more than compensated by the reduction in cache misses. This two phase algorithm is as follows.

	DECst'n 5K/125		DEC 3k/300		HP 9k/720		SUN 10/51	
Algorithm	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
BaseHash	0.465	—	0.096	—	0.277	—	0.171	—
Extraction	0.465	0.0%	0.097	-1.0%	0.282	-1.8%	0.170	0.6%
Clustering	0.402	15.7%	0.088	9.1%	0.268	3.4%	0.153	11.8%

Table 9: Optimizations for the Hash Based Aggregation

Algorithm 4.13 *TwoPhase(R)*

```
ExtractKeyPointer_And_Partition(R)
for each partition i
  Sort(R[i])
  for each t in R[i]
    Initialize/update the group entry for the group
merge the aggregates obtained in each partition
```

In the following section we present some parametric studies we conducted to show that the cache optimizations hold in significantly varying parameter values.

4.3 Parametric Studies

We study the effect of varying relation size, tuple size and the join selectivity on the speedup obtained by the algorithms. In each category of the algorithms, we show how the speedup of the most optimal version over the basic algorithm varies with these changes in parameters and study its implications.

4.3.1 Varying Relation Size

The speedup of the join algorithms, Extraction hash join and ImprovedSort sort-merge join, as a function of relation size is shown in figure 1. We note that whereas the speedup of the MergedStep hash join algorithm decreases slightly with increase in relation size, that of the ImprovedSort sort-merge join improves significantly with increase in relation size. This is important as it shows that cache optimizations are of two kinds:

1. Cache optimizations which depend on the fact that cache would retain a part of the data from previous temporally close accesses. These do not ensure that all the data would be in the cache. Hence, the relative performance gain could decrease with an increase in problem size. These kind of optimizations would benefit from a large cache.

   Examples of this kind are loop fusion, extracting relevant data. Since Extraction hash join uses key pointer extraction and depends on the cache to keep its data, it suffers a little with increase in problem size.

2. Cache optimization which attempt to ensure that cache would retain all the data while it is being processed. This is achieved by ensuring that only that part of the data would be accessed temporally closely which can be retained in the cache. Hence, the relative performance gain would increase with an increase in the problem size. These kind of optimization would not benefit significantly from a larger cache.

   Examples of this kind are partitioning, blocking. Since ImprovedSort uses partitioning its relative performance gain improves with increasing problem size.

4.3.2 Varying Tuple Size

With attribute/pointer extraction, we expect that the performance of the algorithms will not be significantly affected by change in tuple size. Figure 2 shows that increase in tuple size does not significantly affect the speedup of the algorithms.

4.3.3 Varying Join Selectivity

We define join selectivity as $\frac{|Result\ Relation|}{|Inner Relation|}$. We find that varying join selectivity also does not affect the speedup substantially. This is evinced in figure 3.

The results above confirm that the optimizations discussed in the paper are general and hold in substantially differing circumstances.

We finally discuss some issues in result generation.

4.4 Generating The Result Relation in Join Algorithms

There are two main choices regarding result generation in join algorithms.

1. the result tuple is produced on the fly (e.g. as soon as a match is found in case of a join).

2. upon finding a match two pointers to the participating tuples are stored along with a projection list thus generating an in-memory join index [Val87]. Later, depending upon need, the result is generated by accessing the pointed to tuples and doing the projection. This can be considered as lazy evaluation of the result relation.

	DECst'n 5K/125		DEC 3k/300		HP 9k/720		SUN 10/51	
Algorithm	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
BaseSort	2.422	—	0.682	—	1.223	—	0.790	—
TwoPhase	1.754	38.1%	0.396	72.2%	0.854	43.2%	0.667	18.4%

Table 10: Optimizations for the Sort Based Aggregation

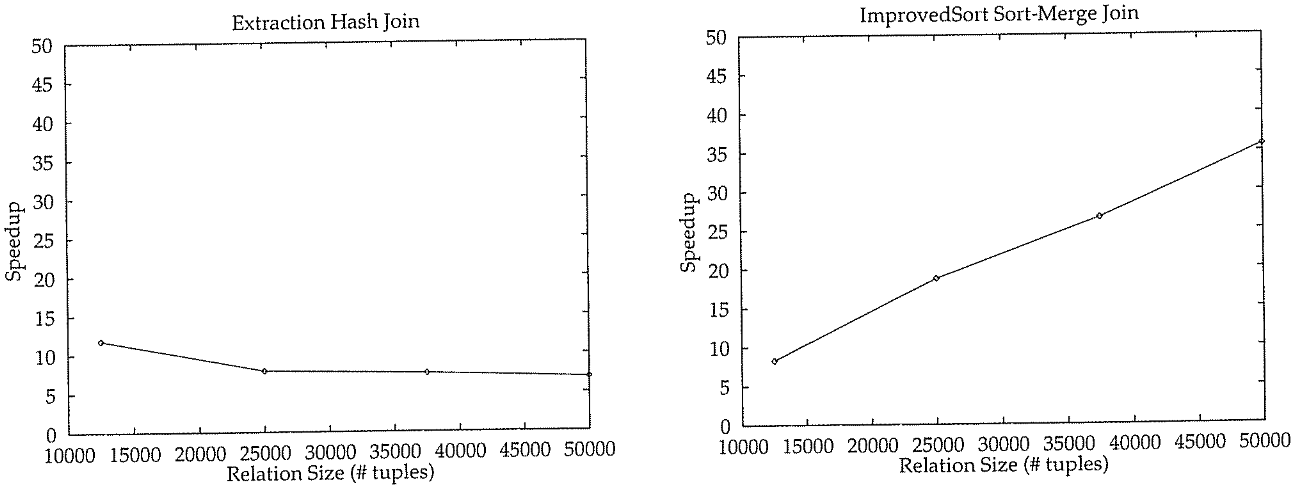

Figure 1: Speedup of Join Algorithms with Varying Relation Size

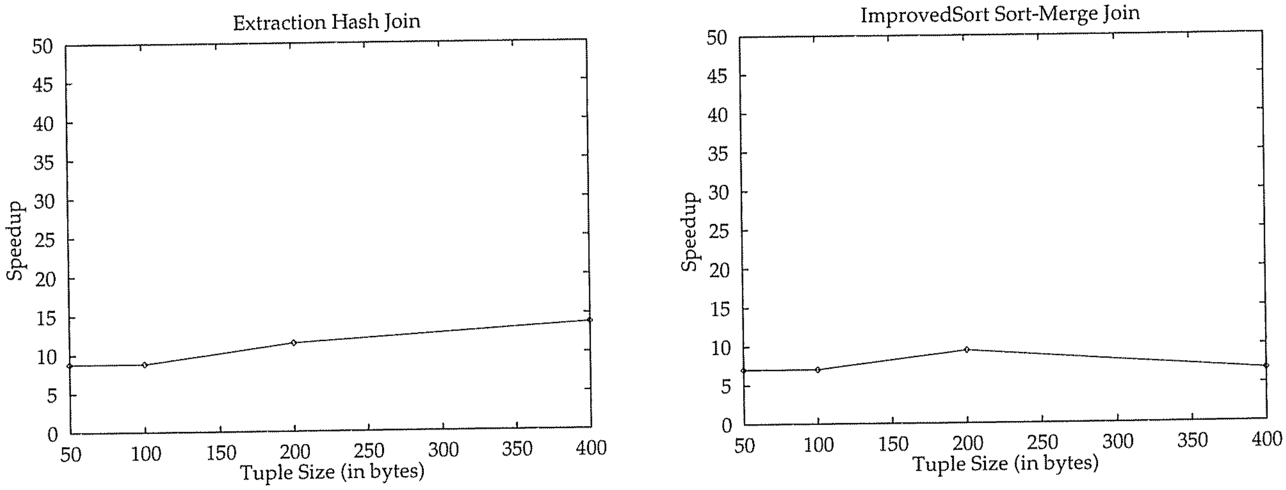

Figure 2: Speedup of Join Algorithms with Varying Tuple Size

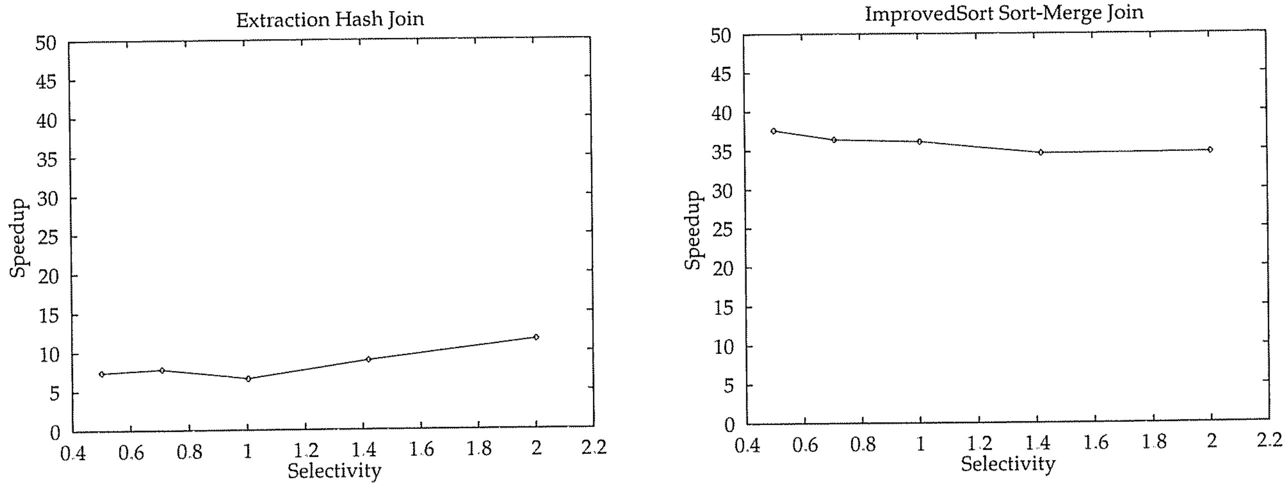

Figure 3: Speedup of Join Algorithm with Varying Join Selectivity

Our experiments show that lazy evaluation of the result relation is almost as fast on the fly evaluation for the algorithms considered. This is because the lazy evaluation, in general, has better cache behavior than on the fly generation. Generating the tuples on the fly results in cache pollution because the generated tuples displace the data required for join processing itself. Lazy evaluation, in contrast, does not pollute the cache but has the overhead of the join index creation resulting in comparable performance of the two approaches. Table 11 shows the above for the DECstation 5000/125 for a few algorithms.

Algorithm	On The Fly	Lazy
Extraction	1.492	1.527
PartitionedHash	1.602	1.527
BaseSortmerge	2.586	2.590
ImprovedSort Sortmerge	2.156	2.132

Table 11: On The Fly vs. Lazy Evaluation

In lazy evaluation, the result generation is independent of the actual computation of the join and therefore it does not affect the performance of the actual join computation. One would expect it to result in a fixed overhead cost dependent only on the characteristics of the result relation (e.g. size of result tuples, size of result relation etc.) and independent of the actual join algorithm used. However our experiments showed this theoretically "fixed" overhead of lazy evaluation is, in practice, slightly variable but always within 5% of each other across the algorithms we studied. This variability is because of the characteristics of the created join index which determines the order in which the tuples will be accessed. In sort merge join, a tuple which is accessed more than once will have all its accesses together. In absence of duplicate accesses, the hash join with its sequential access of the probing relation tuples will show slightly better performance. However, since all these are but secondary effects, their performance impact is not significant.

Furthermore, in main-memory databases, we need not ever generate the final tuples but only access through the resulting pointer structure on the fly. Also, if the buffer pool is large enough to keep all relations participating in a multi-way join then we need not generate complete intermediate result.

For the reasons mentioned above, we decided not to generate the actual result relation. Instead, we left the result in the intermediate pointer format.

5 Conclusions

We have shown that designing algorithms with cache consideration significantly improves their performance. This is most noticeable in the more CPU intensive algorithms, e.g. the nested loop algorithm improves by almost 4 times when we redesign it with the cache in mind. However, much of the time the opportunities for improvement are not evident and one has to use a cache profiler to find the poorly performing parts of the code.

In summary, we have shown that main memory should not be the end of optimization for database algorithms. Designing algorithms that exploit the cache has significant performance dividends and this is becoming increasingly important for the newer generation of microprocessors whose performance critically depend upon effective usage of cache memory. It would be interesting to study how algorithms can exploit specific cache configurations and conversely what cache configurations are better suited for database applications.

Acknowledgements

The authors acknowledge the helpful comments of Paula Hawthorn, Chris Nyberg, and the referees.

References

[BE77] M. W. Blasgen and K. P. Eswaran. Storage and access in relational databases. *IBM Systems Journal*, 16(4), 1977.

[DKO+84] David J. DeWitt, Randy H. Katz, Frank Olken, Lenard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–8, June 1984.

[Eps79] Robert Epstein. Techniques for Processing of Aggregates in Relational Database Systems. Memorandum UCB/ERL M79/8, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, February 1979.

[HS89] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. *IEEE Transacations on Computers*, 38(12):1612–1630, December 1989.

[KH92] R. E. Kessler and Mark D. Hill. Page Placement Algorithms for Real-Indexed Caches. *ACM Transactions in Computer Systems*, 10(4):338–359, November 1992.

[Lar93] James R. Larus. Efficient Program Tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[LW94] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer (to appear)*, June 1994.

[NBC+94] Chris Nyberg, Tom Barclay, Zarca Cvetanovic, Jim Gray, and Dave Lomet. AlphaSort: A RISC Machine Sort. In *Proc. of the 1994 ACM SIGMOD Conf.*, pages 233–242, May 1994.

[Smi82] Alan J. Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.

[Val87] Patrick Valduriez. Join Indices. *ACM Transactions on Database Systems*, 12(2):218 – 246, June 1987.