INformation Systems

Cache-Conscious Radix-Decluster Projections

S. Manegold, P.A. Boncz, N.J. Nes, M.L. Kersten

REPORT INS-E0406 JUNE 2004

ISSN 1386-3681

Cache-Conscious Radix-Decluster Projections

ABSTRACT
As CPUs become more powerful with Moore's law and memory latencies stay constant, the impact of the memory access performance bottleneck continues to grow on relational operators like join, which can exhibit random access on a memory region larger than the hardware caches. While cache-conscious variants for various relational algorithms have been described, previous work has mostly ignored (the cost of) projection columns. However, real-life joins almost always come with projections, such that proper projection column manipulation should be an integral part of any generic join algorithm. In this paper, we analyze cache-conscious hash-join algorithms including projections on two storage schemes: N-ary Storage Model (NSM) and Decomposition Storage Model (DSM). It turns out, that the strategy of first executing the join and only afterwards dealing with the projection columns (i.e., post-projection) on DSM, in combination with a new finely tunable algorithm called Radix-Decluster, outperforms all previously reported projection strategies. To make this result generally applicable, we also outline how DSM Radix-Decluster can be integrated in a NSM-based RDBMS using projection indices.

1998 ACM Computing Classification System: [H.2.4] Main-Memory Database Systems, Query Processing
Keywords and Phrases: main-memory databases; query processing algorithms; cache-conscious algorithms; joins; projections
Note: Work carried out under projects INS1.2 ``Database Architecture'', INS1.3 ``Query Optimizers'', and INS1.4 ``HPQP (MonetDB)''.

Cache-Conscious Radix-Decluster Projections

Stefan Manegold
Stefan.Manegold@cwi.nl

Peter Boncz
Peter.Boncz@cwi.nl

Niels Nes
Niels.Nes@cwi.nl

Martin Kersten
Martin.Kersten@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
As CPUs become more powerful with Moore's law and memory latencies stay constant, the impact of the memory access performance bottleneck continues to grow on relational operators like join, which can exhibit random access on a memory region larger than the hardware caches. While cache-conscious variants for various relational algorithms have been described, previous work has mostly ignored (the cost of) projection columns. However, real-life joins almost always come with projections, such that proper projection column manipulation should be an integral part of any generic join algorithm. In this paper, we analyze cache-conscious hash-join algorithms including projections on two storage schemes: N-ary Storage Model (NSM) and Decomposition Storage Model (DSM). It turns out, that the strategy of first executing the join and only afterwards dealing with the projection columns (i.e., post-projection) on DSM, in combination with a new finely tunable algorithm called *Radix-Decluster*, outperforms all previously reported projection strategies. To make this result generally applicable, we also outline how DSM Radix-Decluster can be integrated in a NSM-based RDBMS using projection indices.

1998 ACM Computing Classification System: [H.2.4] Main-Memory Database Systems, Query Processing
Keywords and Phrases: main-memory databases, query processing algorithms, cache-conscious algorithms
Note: Work carried out under projects INS1.2 “Database Architecture”, INS1.3 “Query Optimizers”, and INS1.4 “HPQP (MonetDB)”.

1. Introduction

Random memory access outside the CPU cache(s) has become very expensive over the past decade and will remain so in the future. As such, the bottleneck for low-level database data access is shifting from I/O to memory access [ADHW99, KPH^{+}98, BGB98]. While the performance penalty for inefficient usage can be dramatic, the database field need not despair. Several decades of progress in database technology has already produced a host of techniques for processing data volumes stored on large but slow memories (i.e., disks) by making efficient use of a smaller but faster memory (RAM). The recent research into *cache-conscious query processing* focuses on transforming these techniques to work one level higher up the memory hierarchy (optimize memory access by making efficient use of the CPU caches) and/or to devise new techniques. We build on recent work into making the join operator cache-conscious, among others by introducing a *Partitioned Hash-Join* [SKN94] that can be paired with a fine-grained partitioning operator called *Radix-Cluster* [BMK99] to partition huge relations into a large number of small clusters that each fit a CPU cache with just a few tens of KBs.

A limitation of these previous efforts is that so far they only considered joins on thin relations consisting solely of the join keys and producing only a table of matching `oid` pairs (i.e., a join-index [Val87]). However, any real-life RDBMS join query goes accompanied by some projection of non-join columns into the result. The cost of handling such projection columns depends on their number, type(s) and the relation cardinalities (both inputs and result). The actual cost impact can vary from zero (in the not-so-realistic case where there are no projections at all), to totally dominating (e.g., imagine a join with thousands of projection columns to propagate feature vectors in a multimedia application). In our performance evaluation, we find that queries may spend more than 90% of their time in projection. Therefore, efficient handling of projections should be part of any cache-conscious join technique.

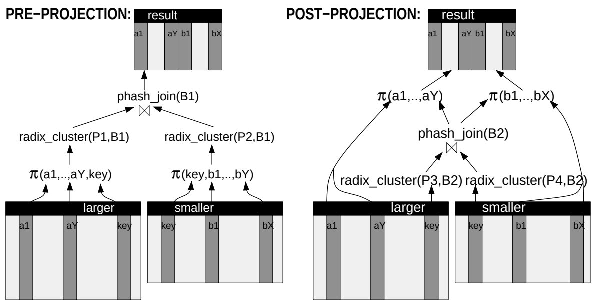


Figure 1: Pre- vs. Post-Projection

1.1 Problem Statement

This paper describes optimization of CPU- and memory-resources of generic equi-join *including* projections:

```
SELECT larger.a1, .., larger.aY,
       smaller.b1, .., smaller.bZ
FROM   larger, smaller
WHERE  larger.key = smaller.key
```

The focal point of our analysis is the performance impact of the amount of projection columns `a1..aY` respectively `b1..bZ`, given various relation and join result sizes. Handling projections efficiently only becomes hard when *both* the `smaller` and `larger` table have many tuples, such that their individual columns do not fit the cache. Our *Radix-Decluster* algorithm addresses this situation.

The commonly applied projection strategy in a RDBMS is *pre-projection* (see Figure 1), where the projection columns are fetched in the table scans preceding the join, and where the projection column values travel as 'extra luggage' together with the join keys through the join pipeline. In contrast, Radix-Decluster is a *post-projection* method, i.e., one where first the join result is computed, creating a (partial) join-index, and only afterwards the full query result is produced by computing the projection columns. Though we focus the experiments on one particular join algorithm (Partitioned Hash-Join), the Radix-Decluster algorithm is independent of the join method chosen.

RAM vs. Disk Optimization Since we have already mentioned the analogy between optimizing CPU cache-access and optimizing disk access, it is instructive to point out the main similarities and differences. As for similarities, both disk and RAM have to contend with a high random access latency, that relative to CPU speed is increasing exponentially over time. Also just like disk, RAM is a block device (block=cache line), and sequential data access has now become much faster than random access, even when random access makes use of *all* data in the block (we call this “optimal” random access). This effect is caused by a new feature in memory subsystems called *data prefetching*: the CPU or in some cases the memory chipset automatically detect sequential access patterns and schedule data loads in advance for these [LH99, HSU^{+}01]. This is complemented by advances in DRAM technology, which keeps banks of recently accessed locations open, such that adjacent locations can be more quickly available. On our experimentation platform, sequential access – as obtained by STREAM [McC95] – is almost 10 times faster than “optimal” random access: 3.2GB/s vs. 360MB/s (a 178ns latency for getting a cache-line of 64 bytes makes for 360MB/s).

An important difference between disk and RAM is that the disk can be controlled using an OS interface, allowing traditional DBMS systems full control over their buffer cache. In contrast, RAM is cached implicitly in hardware, (most often) using an LRU mechanism with limited associativity. Thus, the only way that query processing algorithms can now influence RAM caching is indirectly by controlling data placement and access pattern. A second difference is the small granularity of the CPU caches. There is a “15 year gap” between CPU cache and RAM sizes: problem sizes of 2004 must now be crammed in caches having the RAM sizes of 1989. This means that e.g., partitioning to fit something large into the CPU cache must create *many* more small partitions than classical partitioning to fit something on disk into RAM ever had to. Having to manage (tens of) thousands of partitions rather than a handful can expose bottlenecks that remained unnoticed in the disk case, as we will see in our discussion of the Radix-Cluster algorithm.

Experimentation Platform The work reported here partly builds upon the research into cache-conscious query processing in the MonetDB project. MonetDB[1] is a main-memory database system targeted at query-intensive applications [Bon02] that uses a vertically fragmented storage scheme called the Decomposition Storage Model (DSM) [CK85]. In DSM, each tuple gets a unique system-generated `oid` that is typically densely ascending (0,1,2,...), and for each column a DSM table is created that holds `[oid,value]` pairs. Comparable to what RowIds are in Oracle, the MonetDB system has support for *implicit* columns – also dubbed `void` columns ("virtual-oids") – to represent such densely ascending `oid` columns on the logical level without taking any physical storage. Thus in MonetDB, each relational column is stored in a separate `[void,value]` table. Most DSM systems [Syb96, RDS02] do away with the extra storage for the `oid`s, such that the DSM data layout boils down to a single array for each column. DSM is cache-friendly when (OLAP) queries need only a subset of all table columns (i.e., in case of low *projectivity*). In the commonly used NSM storage scheme (i.e., a layout with each tuple contiguously stored), this means that parts of the cache line will not be used. In DSM, each cache line only contains values from the same column, and only relevant columns are loaded, achieving optimal cache line usage.

A second characteristic of MonetDB is its column-wise query processing model, which allowed for an implementation of its query processing algebra without need for an interpreter to evaluate expressions (each operation performs a simple, hard-coded, operation on large arrays of values, producing a new column as result). This goes in conjunction with the absence of low-level record/attribute lookup and data movement functionality, as columns are accessible by position as arrays of a homogeneous type. The experiments performed confirm these factors give MonetDB a significant advantage in terms of raw CPU efficiency that is strongly linked to this query execution model.

The third main characteristic of MonetDB is cache-conscious query processing. MonetDB has been the birth ground for a number of novel cache-conscious algorithms [BMK99]. *Radix-Decluster* – the contribution of this paper – is a crucial addition to this collection.

Related Work Though our experimentation platform is MonetDB, which is a DSM system, we compare our approach with its more common counterpart NSM, and in particular with pre-projection in NSM (which is used in almost all commercial database systems). However, there has recently also been some research into NSM post-projection, in particular the Slam- and Jive-Join algorithms [LR99]. While these algorithms work under the assumption that the join-index is already computed and available (hence pre-projection is not an option), and they are designed mainly for an I/O setting, we also include them in our NSM comparison with Radix-Decluster to evaluate their usefulness from the perspective of cache-conscious query processing.

An interesting alternative storage scheme is PAX[ADHS01], which basically does DSM within an NSM disk page. Thus, PAX cache-line usage can be as efficient as DSM under low projectivity, but PAX still wastes I/O bandwidth on such queries, which easily can cause a performance bottleneck. Though we will make our case that Radix-Decluster on DSM can be scaled to a disk-based RDBMS that runs on a high bandwidth I/O subsystem (e.g., using a well-sized RAID array of SCSI disks controlled through PCI-X), our experimentation is limited to main-memory execution, by lack of such an (expensive) setup. As in main-memory the difference between PAX and DSM is small, we limit ourselves here to the two extremes NSM and DSM.

Finally, we build here on previous work on detailed performance modeling of hierarchical memory access cost [Man02, MBK02] using hardware-independent formulas that are parametrized by all relevant architectural characteristics. These parameters can be derived automatically at run-time with the `Calibrator` utility [2], which is also integrated in MonetDB. The cost formulas are easy-to-define as they consist of a combination of a number of basic patterns (with known formulas) that can be combined automatically with composition

[1] Available at `http://www.sourceforge.net/projects/monetdb`

[2] Calibrator is available from `http://monetdb.cwi.nl/Calibrator`

functions. In all, these cost models allow us to quickly analyze the behavior of the various algorithms, and to draw conclusions on their optimal parameter settings.

1.2 Outline

In Section 2, we give a short re-cap on cache-conscious Partitioned Hash-Join and Radix-Cluster, which are basic building blocks in this research. In Section 3, we show how Radix-Cluster can be used to optimize memory access of post-projections to one of the join relations. In order to optimize cache usage for projections on *both* join relations, we then introduce our new *Radix-Decluster* algorithm. In Section 4, we perform exhaustive experiments with pre- and post-projection strategies both for the DSM and NSM storage schemes, and compare non cache-optimized strategies with our Radix algorithms, as well as with Jive-Join. In Section 5 we make our case why and how DSM post-projection with Radix-Decluster should be integrated in standard RDBMS technology, before we present our conclusions and discuss directions for future work in Section 6.

2. CACHE-CONSCIOUS JOIN

We give a short re-cap on cache-conscious join, using *Partitioned Hash-Join* in conjunction with *Radix-Cluster* [BMK99]. In Appendix A, we give cost model descriptions for these algorithms, and show how these correctly predict their performance (see resp. Figures 9a and 9b).

2.1 Partitioned Hash-Join

In the Hash-Join algorithm considered in this paper, the outer relation is scanned sequentially, while a hash-table is used to probe the inner relation. The very nature of the hashing algorithm implies that the access pattern to the inner relation (plus hash-table) is random. Therefore, *Partitioned Hash-Join* first scans both relations, and partitions them according to a hashing criterion, making each inner partition smaller than the cache size, such that the subsequent Hash-Joins on the corresponding partitions all have good cache behavior [SKN94]. The "cursors" in the output partitions where the partitioning operator inserts tuples as it scans its input, all need to be in a cache-line in order to achieve good performance during partitioning. As the number of available cache lines is limited (especially in systems that have a slow TLB cache, with usually only 64 entries) and the number of cursors grows with the size of the relation (a bigger relation leads to more partitions of a given size), the simple single-pass partitioning is limited in its scalability: above a certain relation size, the partitioning operation itself becomes a performance problem due to cache thrashing, as not all cursors can be kept in cache anymore.

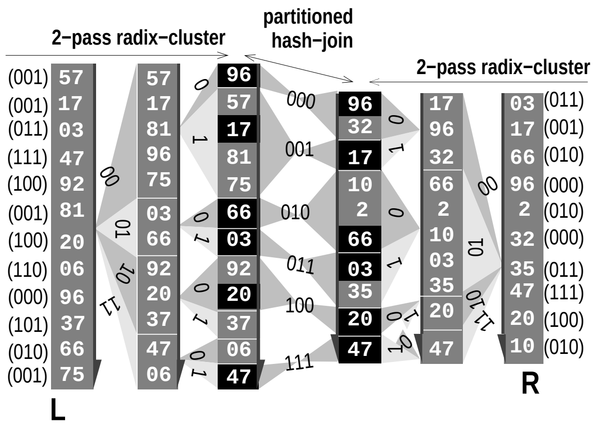
2.2 Radix-Cluster

The *Radix-Cluster* algorithm, which uses incremental multi-pass partitioning, has been shown to solve the operand partitioning problem. It provides efficient partitionings needed for large joins in two or even more passes [BMK99]. Briefly, `radix_cluster`(B,P) uses the lower B *Radix-Bits* of the integer hash-value of the join attribute to cluster a relation into $H = 2^B$ partitions. By performing P sequential passes, each of which use B_p bits, starting from the left ($\sum_1^P B_p = B$), Radix-Cluster limits the number of partitions created per pass to $H_p = 2^{B_p}$ ($\prod_1^P H_p = H$). Figure 2 sketches a Partitioned Hash-Join of two relations L and R. First, both relations are clustered into 8 partitions (3 bits) using 2 passes. The first pass uses the 2 left-most of the lower 3 bits to create 4 partitions. In the second pass, each of these partitions is sub-divided into 2 partitions using the remaining bit. Once both relations are clustered, a hash-join is performed on all matching partitions. For ease of presentation, we did not apply a hash-function in Figure 2. In practice, though, a hash function should even be used on integer values to ensure that all bits of the join attribute play a role in the lower B bits used for clustering.

3. DSM POST-PROJECTION

The DSM post-projection strategy has two phases:

1. *Make a join-index.* First we access only the DSM tables storing the key columns, and join these together to find matching pairs of tuples: a join-index [Val87].



black tuples hit (lowest 3-bits of values in parenthesis)

Figure 2: Partitioned Hash-join

2. *Do column projections.* One-by-one, we construct the columns of the result relation, each in a separate DSM table, by using the join-index to fetch values from one input column (also stored in a DSM table).

The join-index consists of `[oid,oid]` combinations of pointers into both "smaller" and "larger" input relations. These `oid`s are not necessarily implemented as pointers, but may also be integer record numbers, byte offsets or RowIds (combinations of disk-block numbers and byte offsets). The projection operations are Pointer-Based Joins or *Positional-Joins*, with negligible CPU cost. In MonetDB, columns are stored in `[void,value]` tables, which are implement as arrays [3]. Thus, an `oid` is a simple integer (starting at 0 for the first entry), and Positional-Join equals array lookup.

One should note that the DSM post-projection join strategy *materializes* the join result. This is inevitable for the so-called "hard" join cases, where we must join two relations that do not fit the small-but-fast memory (i.e., the CPU cache). This is similar to scalable I/O-based join algorithms such as Sort-Merge-Join or Hybrid Hash-Join, that must be applied when the inner relation exceeds the RAM buffer size and pipelining is not possible.

In MonetDB, a join only is "hard" if the *individual* columns - rather than the entire "smaller" relation - exceed the CPU cache. In the other, so-called "easy" cases, we can use e.g., simple non-partitioned Hash-Join, by building a hash-table on the "smaller" relation to generate the join-index. The join-index will then contain the `oid`s of the "larger" relation in ascending order, such that the Positional-Joins for projecting the input columns into the result exhibit a sequential RAM access pattern. As discussed in Section 1.1, sequential RAM access is well-supported by modern hardware. In contrast, the Positional-Joins for the projections from the "smaller" relation will have a random access pattern. Luckily, these columns fit the CPU cache in the "easy" cases, so the cache-lines where the input columns are stored will stay cached in the CPU after the first access, such that subsequent (adjacent) data fetches can be serviced from the cache.

In this paper, we attack the problem of executing "hard" joins in a cache-conscious manner. With CPU caches limited to a couple of MBs, and assuming an average column-width of 4 bytes, this currently translates into joins between (intermediate) relations that *both* have 500K or more tuples, which is a common and thus relevant problem.

3.1 Partial Radix-Cluster

We use Partitioned Hash-join, as described in Section 2, to join two relations that both exceed the CPU cache in a cache-conscious manner. Due to the nature of Partitioned Hash-Join, neither the `oid`s of the "larger" nor

[3]Columns of variable-sized types like string use an extra – separate – memory buffer, where the array simply contains integer offsets into.

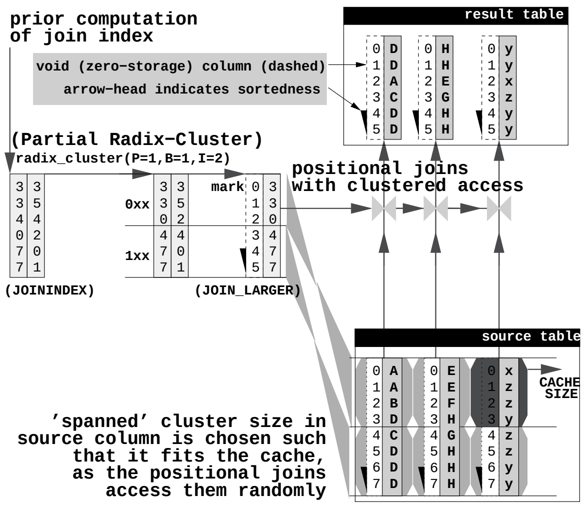

prior computation of join index
void (zero-storage) column (dashed)
arrow-head indicates sortedness
result table
(Partial Radix-Cluster)
radix_cluster(P=1,B=1,I=2)
positional joins with clustered access
0xx
1xx
mark
(JOININDEX)
(JOIN_LARGER)
source table
CACHE SIZE
'spanned' cluster size in source column is chosen such that it fits the cache, as the positional joins access them randomly

Figure 3: Projection Joins Using Partial Radix-Cluster

those of the "smaller" relation appear in ascending order in the resulting join-index. A (standard) improvement is therefore to sort the join-index, in the order of the oids of the "larger" relation. In MonetDB, we re-use the Radix-Cluster algorithm as a fast *Radix-Sort*, by exploiting the property that oids stem from dense domains $[0..N\rangle$ (where N is the size of some relation). For all types but oid, Radix-Cluster transforms each value with a hash function, both to obtain integer bits and to combat skew. For oids, hashing is not applied as oids are integers already and not skewed. This also means that a Radix-Cluster on all *significant* bits (i.e., the lowermost $log_2(N)$ bits) is equivalent to Radix-Sort. Radix-Sort can be compared with traditional run-generating sort algorithms, as it also partitions the data on a sequential pass, and then (iteratively) further processes each partition.

Fully sorting the join-index, however, is overkill as a *partial ordering* can achieve the same effect. If the join-index consists of clusters that each contain oids of only a certain (disjoint) range, a Positional-Join into a projection column sequentially processes each cluster one by one, and while processing each individual cluster, accesses only a limited region in the projection column. If this region is small enough (such that it fits the cache), the algorithm will approach optimal cache (re-)usage. To make partial clustering possible, we added the possibility to indicate to Radix-Cluster to stop early and ignore a certain number of lower Radix-Bits. Stopping early leaves the relation unsorted on the lowermost bits (i.e., partially ordered). The benefit of this "partial-cluster" strategy is that it has the potential to optimize memory performance of the column projections using Positional-Joins just as well as a full Radix-Sort, but at a clustering cost that is much less.

Figure 3 shows that we first Partially Radix-Cluster JOININDEX in one pass ($P = 1$), using one Radix-Bit ($B = 1$) and stopping early at the first ($I = 1$), lowermost, Radix-Bit. On the resulting [oid,oid] table, we create a [void,oid] view JOIN_LARGER (using the mark() operator [Bon02]). The right column of JOIN_LARGER contains the clustered oid column, and the left column consist of a new densely ascending oid sequence that represents the join result. Subsequent Positional-Joins between this JOIN_LARGER view and the input columns have a nice sequential access pattern, eliminating the cache problem. We compute the optimal number of Radix-Bits B and Ignore-Bits I as follows:

$$B = 1 + log_2(|\textit{COLUMN}|) - log_2(C\,/\,\overline{\underline{\textit{COLUMN}}})$$
$$I = log_2(|\textit{JOININDEX}|) - B$$

where $|R|$ denotes the number of tuples in a table R, $\overline{\underline{R}}$ denotes the byte-width of these tuples, C is the size of the cache in bytes (see [Man02, MBK02]). For example, if we have a CPU cache of 64KB and we have values that are 4 bytes wide, then a cluster of 16,384 tuples would just fit. If the source table from where the projections come has 10M tuples, we would create 2^{10} = 1024 clusters to arrive at a mean cluster size of 10,000 (which

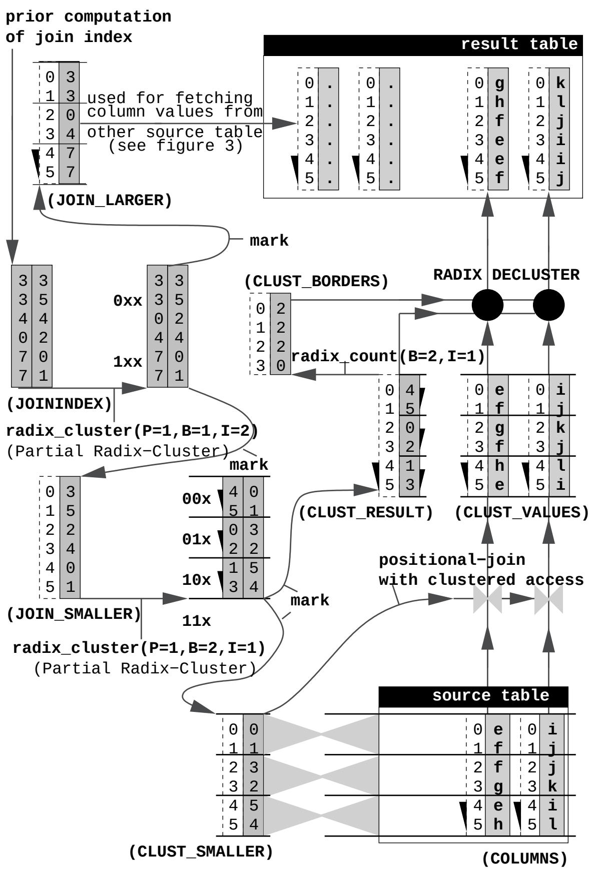


Figure 4: Optimized DSM Post-Projection Using Radix-Decluster

would be the largest cluster size $<$ 16,384). Such clusters can be created with a partial Radix-Cluster on the highest significant 10 bits (i.e., bits 24-15, as log_2(10M) = 24), allowing Radix-Sort to ignore the lowermost 14 bits.

3.2 Radix-Decluster

Even when using Partial Radix-Cluster to optimize projections into the “larger” relation, cache problems still occur for the projections from the “smaller” relation. It is clear that the join-index (and thus the join result) cannot simultaneously be clustered in *both* oid orders. Figure 4 shows that after performing the projections into the “larger” relation, we re-cluster the view JOIN_SMALLER (that similar to JOIN_LARGER consists of fresh densely ascending oids left, paired with the right column of the clustered join-index). This yields a temporary [oid,oid] table. We then create two [void,oid] views CLUST_RESULT and CLUST_SMALLER from this table using the mark() operator. The left column of these views is a fresh “void” column of new ascending oids. The right column of CLUST_SMALLER holds the oids of the join-index that point into the “smaller” table in a nice clustered order, while the corresponding values of the right column of CLUST_RESULT hold the correct position of those join-tuples in the final result. The next step in the process is to use CLUST_SMALLER to perform the projections with cache-efficient Positional-Joins. This, however, produces projection columns (denoted CLUST_VALUES) which are not yet in the correct order. The *Radix-Decluster* algorithm – depicted in detail in Figure 5 – performs the task of putting them in the correct final result order in a cache-friendly manner.

Radix-Decluster exploits the following two properties of the right column of CLUST_RESULT, which was created by Radix-Clustering a left void column on the order of its right column: (1) as Radix-Cluster neither adds

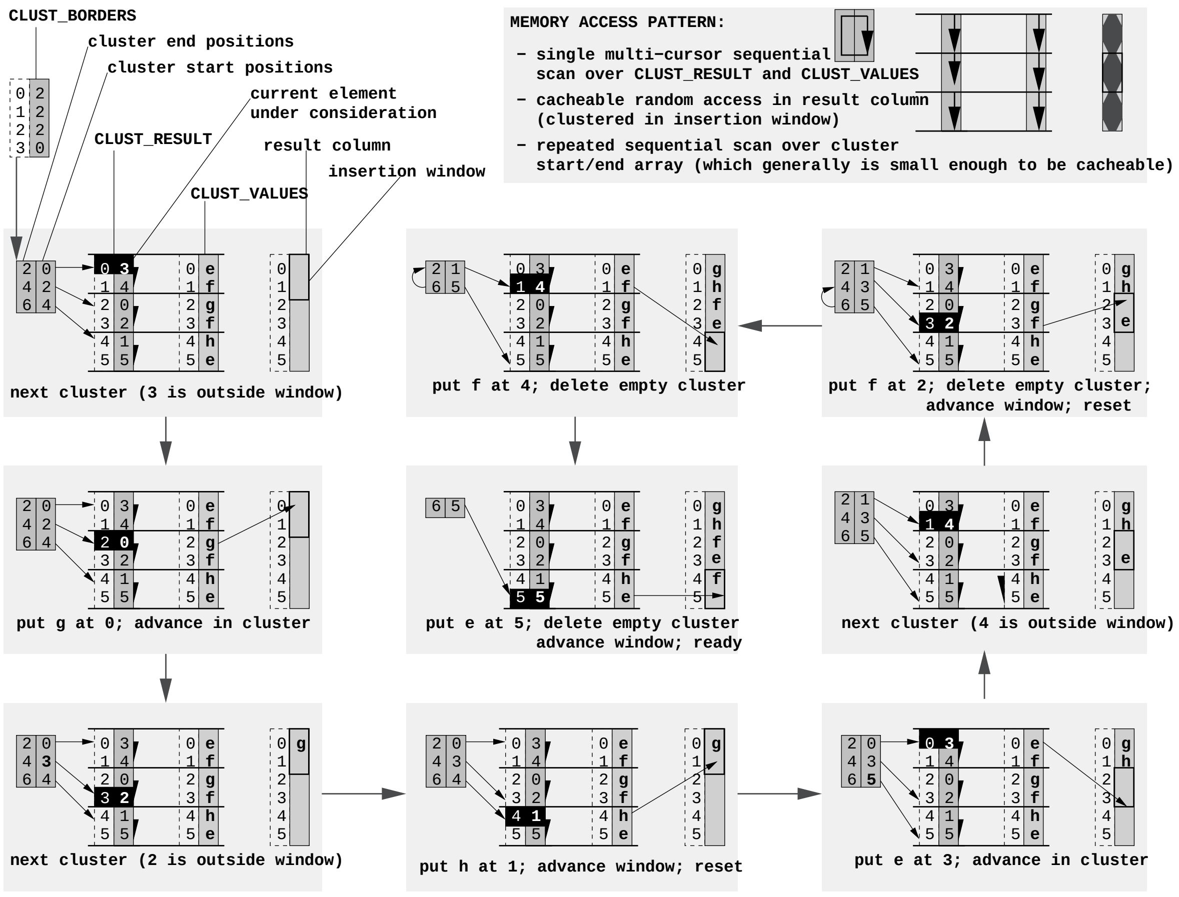

CLUST_BORDERS
cluster end positions
cluster start positions
current element under consideration
CLUST_RESULT
result column
insertion window
CLUST_VALUES
MEMORY ACCESS PATTERN:
– single multi-cursor sequential scan over CLUST_RESULT and CLUST_VALUES
– cacheable random access in result column (clustered in insertion window)
– repeated sequential scan over cluster start/end array (which generally is small enough to be cacheable)
next cluster (3 is outside window)
put g at 0; advance in cluster
next cluster (2 is outside window)
put h at 1; advance window; reset
put e at 3; advance in cluster
next cluster (4 is outside window)
put f at 2; delete empty cluster; advance window; reset
put f at 4; delete empty cluster
put e at 5; delete empty cluster advance window; ready

Figure 5: The Memory Access Pattern Of Radix-Decluster

nor deletes any values, this column would again form a dense sequence $(0,1,..N-1)$ when sorted. (2) within each cluster, the oids are still sorted. This happens because Radix-Cluster scans its input sequentially, and appends values to their respective output cluster, thus locally respecting the input order.

Property (2) implies that this right column can be sorted by **merging** all sorted clusters. However, the CPU cost of a merge of N tuples partitioned over $H = 2^B$ sorted clusters is at least $O(log(H)N)$. Alternatively, using Property (1) we could just insert the values from CLUST_VALUES in the result array using the oids from CLUST_RESULT as array index, with CPU cost $O(N)$. However, these insertions would constitute a random access pattern larger than the CPU cache.

We obtain the best of both approaches, by restricting the random access to an *insertion-window* W (cf., Figure 5). Each iteration of the algorithm processes each cluster once, advancing a cursor in it while the oids still fit in the window, inserting the values at this oid position. Property (1) tells that after processing each cluster once, *all* positions in the insertion window will have been filled (it is a dense oid sequence). Then, the window is shifted $|W|$ positions and the process repeats until all cursors have reached the end of their cluster. The window size $|W|$ is preferably much larger than the number of clusters, such that per iteration in each cluster multiple tuples fall into the window. These multiple tuples are accessed sequentially in both CLUST_RESULT and CLUST_VALUES. This memory access pattern is crucial, as the sequential access fully uses the cache lines that store both columns. The only restriction is that $|W|$ must fit the memory cache (i.e., $||W|| \leq C$), as it is filled in random order.

Pseudo-code of the algorithm is in Figure 6. The radix_count previously mentioned in Figure 4, analyzes

```
<Type>[]
radix_decluster<Type>(
  int                       cardinality, nclusters,
  Type                      values[cardinality],
  oid                       IDs[cardinality],
  struct { int start, end } cluster[nclusters])
{
  <Type> result_column[] = malloc(cardinality*sizeof(<Type>));
  int windowLimit, windowSize = CACHESIZE / 2*sizeof(<Type>);

  for(windowLimit=windowSize; nclusters>0; windowLimit+=windowSize) {
    for(int i=0; i < nclusters; i++) {
      while (IDs[cluster[i].start] < windowLimit) {
        result_column[IDs[cluster[i].start]] = values[cluster[i].start];
        if (++cluster[i].start >= cluster[i].end) {
          cluster[i] = cluster[--nclusters]; // delete empty cluster
          if (i >= nclusters) break;
        }
      } // while more cluster elements in window
    } // while more clusters to merge
  } // while more insertion windows to fill result
  return result_column;
}
```

Figure 6: The Radix-Decluster Algorithm

a (partially) Radix-Clustered column and returns the actual sizes of the clusters. These sizes are used in the Radix-Decluster to initialize the `cluster` border structure.

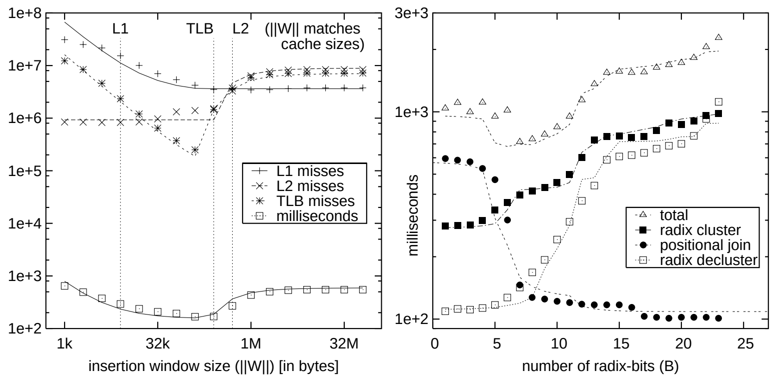
The Radix-Decluster projection strategy is more expensive than the partial-cluster strategy discussed earlier. Both strategies feature one initial Radix-Cluster, and for each projection column a Positional-Join, but the former adds an extra Radix-Decluster operation for each projection column. Hence, it will only be used for getting projection columns from the table with cheaper projections. Which input relation in the join has the cheapest projection phase depends on the number of projection columns in both relations, the data types in these projection columns, and the number of tuples in both input relations.

4. Performance Evaluation

In this section, we present experiments done on a 2.2GHz Pentium 4 machine, with a 64-entry TLB with miss latency of 50 cycles, a 16KB L1 cache with 32-byte cache lines and a miss latency of 28 cycles, a 512KB L2 cache with 128-byte lines and a miss latency of 350 cycles (i.e, the latency of the 2GB PC800 RDRAM main memory is 178ns).[4] Our experimentation platform is MonetDB, also in the NSM experiments, where NSM is "simulated" by introducing new atomic types that hold 1, 4, 16, 64, and 256 integer column values, and which are copied and projected from using a NSM projection routine that iterates over such a "record" and copies selected values out of it.

In our experiments, we executed our example project-join SQL query using various DSM and NSM query processing strategies described in the following. We use relations of equal size N ranging from 15K to 16M tuples, consisting of $\omega \in \{1,4,16,64\}$ all-integer (4-byte) columns. We vary the join hit rates $h \in \{3,1,0.3\}$, and project $\pi \in \{1,4,16,64|\pi \leq \omega\}$ columns from both relations into the result. Finally, we also present exper-

[4]The early work on cache-conscious query processing [SKN94] reported a 30 cycle latency, thus we observe a 12-fold increase in 9 years.

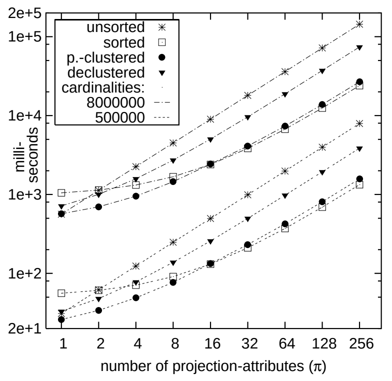


Modeled (lines) vs. Measured (points) Performance

a) Number of Events and Elapsed Time (input clustered on 8 bits)

b) Components and Total Cost (using best insertion window size)

Figure 7: Radix-Decluster ($N = 8$M, $\pi = 1$)



"larger" table: Unsorted vs Sorted vs Radix-Cluster
"smaller" table: Unsorted vs Radix-Decluster
(see text for details)

Figure 8: DSM Post-Projection

iments where one of the join relations is a selection on a base-table that selected a fraction $s \in \{1, 0.1, 0.01\}$, such that we get *sparse* projections. In all experiments, all processing happens in main-memory (no I/O or page faults).

4.1 DSM Post-Projection Experiments

We first analyze the performance behavior of Radix-Decluster in isolation. Figure 7a shows the relationship between size of the insertion window (cf., Section 3.2 and Figure 5) and performance. We used hardware performance counters [HSU$^+$01] to obtain detailed information on the amount of L1, L2 and TLB misses. This data enabled us to formulate and validate the performance model described in Appendix A. In this formula, $\#w = |X'|/|W|$ denotes the total number of insertion windows used. Our models can predict and accurately explain what is happening, as is seen by the fact that the dots (values obtained by experiments) and lines (the cost model) in Figures 7a, 7b and 9d nicely coincide.

If we look in detail at Figure 7, we see Radix-Decluster become faster as the insertion window becomes larger, which is explained by the fact that a larger insertion window leads to higher average number of tuples w processed per cluster in each iteration, improving sequential memory bandwidth usage in `CLUST_RESULT` and `CLUST_VALUES`. However, the insertion window sustains a random access pattern, such that when $||W||$ becomes bigger than the cache size C (our L2 has 512KB), performance drops sharply, due to an increase in L2 misses. A less important threshold is when $||W||$ becomes bigger than the number of pages that fit the TLB, after which TLB misses will start to occur during the inserts. Both these thresholds are drawn in Figure 7a. Another cause for TLB misses is the number of input clusters: if it is bigger than the number of TLB entries (and it is, in the depicted case of 8 Radix-Bits = 256 clusters), each Radix-Decluster iteration will cause two TLB misses when starting to process a new cluster, both in `CLUST_RESULT` and `CLUST_VALUES`. However, this happens only every one in w tuples, such that its impact diminishes quickly with increasing window size. Our analysis showed that choosing $w = 32$ is sufficient to achieve good memory bandwidth usage, and this is the value we use in Figure 9d to confirm the accuracy of our model on multiple cardinalities and Radix-Bits.

We then turn our attention to the interplay between Radix-Cluster, Positional-Join and Radix-Decluster in our Radix-Decluster DSM post-projection strategy, as depicted in Figure 7b. In Section 3.1, we already gave a formula for computing a good number of bits for Radix-Clustering the join-index, such that the subsequent Positional-Joins run well. Figure 9c confirms the accuracy of our predictive model for Positional-Joins between relations of multiple cardinalities (hit-rate 1), clustered with varying granularity (Radix-Bits). In the setting of Figure 7b we can indeed verify that $|R| = 8$M, $\overline{\underline{R}} = 4$ leads to $B = 8$, which is the lowest number of Radix-Bits for which Positional-Join runs optimally (it then achieves minimal L2 misses). This is usually the optimal point

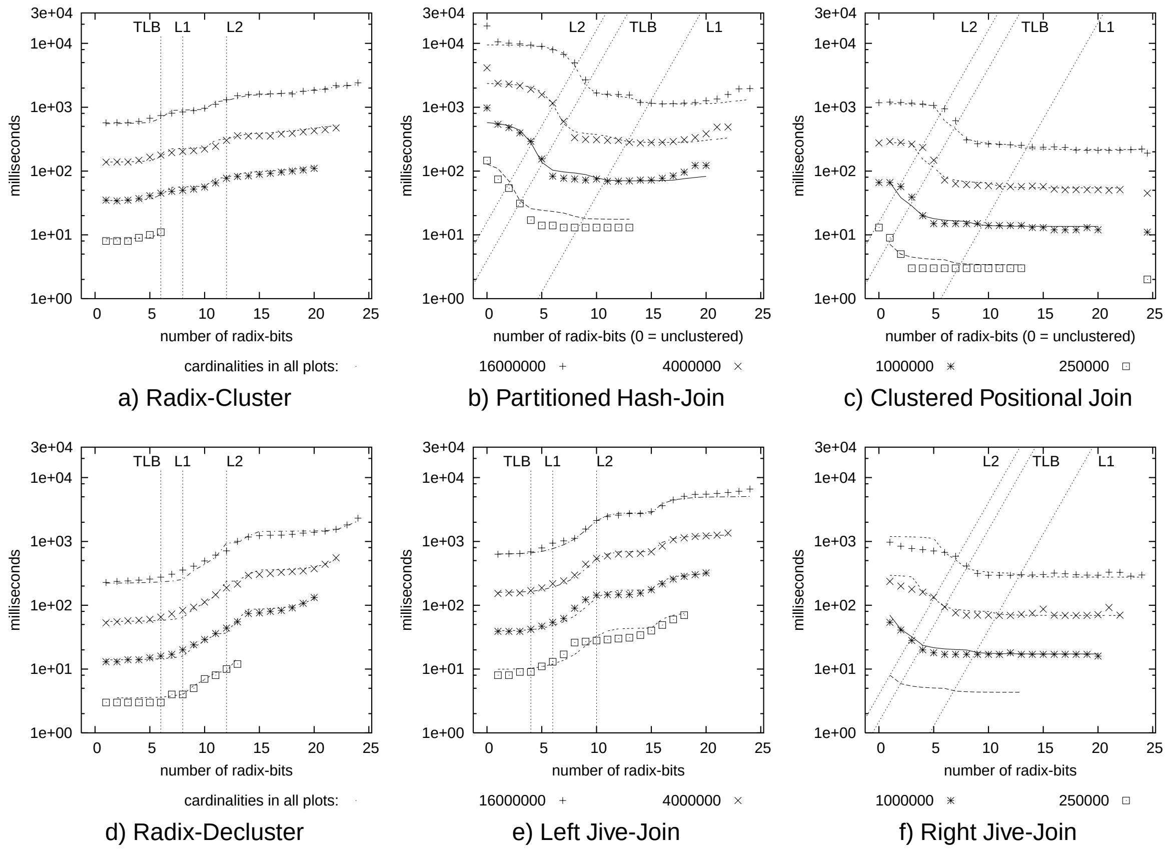
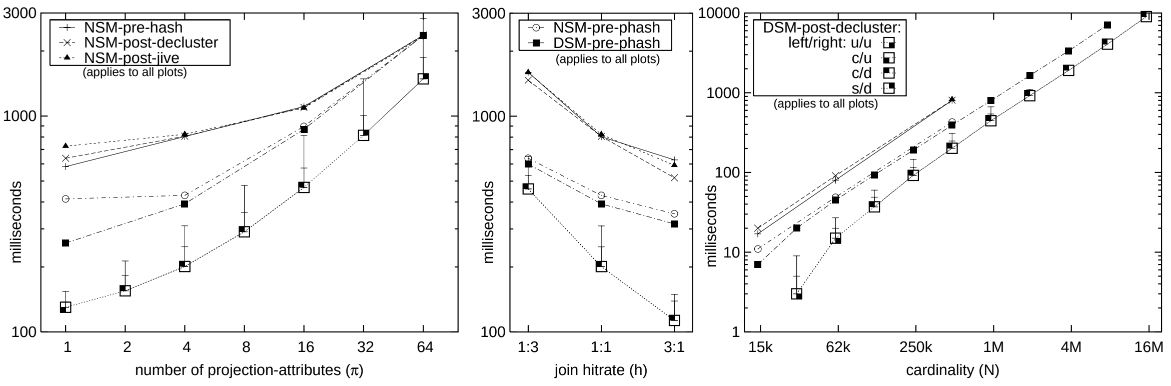


Figure 9: Modeled (lines) vs. Measured (points) Performance of various Join-Phases (DSM, $\pi = 1$)

overall, as Radix-Decluster cost only increases with more Radix-Bits. It sometimes is better to use even fewer Radix-Bits. The performance hit taken on Positional-Join, might then be compensated by a cheaper Radix-Cluster. As Radix-Cluster is executed only once, but Positional-Join for every projection column, this usually happens only if the number of projection columns π is very low. To perform well, that is, without running into cache or TLB problems, Radix-Decluster is limited by two factors. First, we need to process a sufficiently high w tuples from each input cluster to exploit the sequential memory bandwidth. We saw above, that $w = 32$ is the value to choose. Second, the insertion window size must not exceed the cache size C. From this, we can conclude that Radix-Decluster can handle relations of sizes up to $|R| = C^2/(32 * \underline{\overline{W}}^2)$ efficiently. This formula resembles a similar bound as given in [LR99] for Jive-Join.

We finally analyze which DSM post-projection strategy for our generic join query works best and under which circumstances. Note that for DSM systems only π matters, not the actual number of columns in the table ω (as they are fragmented vertically in distinct columns - and the unused columns stay untouched). Therefore, a DSM experiment for a certain π holds for all ω. We consider four strategies, each identified with a one-letter code:

- u *Unsorted:* one Positional-Join from the join-index into each projection column.
- s *Sorted:* first Radix-Sort the join-index, then execute the Positional-Joins.
- c *partial-Cluster:* first partially cluster the join-index. We take the number of Radix-Bits that works best (on our platform, this leads to 256KB clusters).
- d *radix-Decluster:* like the previous, but each Positional-Join is followed by Radix-Decluster.



a) Impact of Projectivity ($N = 500\text{K}$, $\omega = 64$, $h = 1{:}1$)

b) Impact of Join Hit Rate ($N = 500\text{K}$, $\omega = 64$, $\pi = 4$)

c) Impact of Cardinality ($\omega = 64$, $\pi = 4$, $h = 1{:}1$)

DSM Post-Proj. vs NSM Post-Proj. (Radix-Decluster & Jive-Join) vs NSM Pre-Proj. (simple & partitioned Hash-Join) vs DSM Pre-Proj.
(Error bars indicate *sparse* DSM Post-Projection performance; i.e., one join relation is a 10% resp. 1% selection of a larger base table.)

Figure 10: Overall Join Performance

Figure 8 summarizes the performance of the various DSM post-projection strategies, depending on the amount of projection columns π and cardinality N. For small cardinalities ($N \leq 125\text{K}$), all strategies that do any kind of reordering lose to simple unsorted processing of the Positional-Joins, since the columns are so small that they fit the cache anyway. For larger cardinalities, however, the unsorted approach always loses by a big margin (e.g., by almost a factor 10 at $N = 8\text{M}$ and $\pi = 256$). With small π, partial-clustered processing beats sorted processing. The gap shrinks with growing π, and with $\pi > 16$, sorted processing wins. Finally, we see that the Radix-Decluster strategy always loses from the partial-cluster strategy, but is actually quite competitive, beating unsorted processing by a large margin. As explained, Radix-Decluster is to be used only for the second (smaller) projection table, with unsorted processing as the only alternative, as sorting or partial-cluster is only applicable to the first projection table.

4.2 Comparison of Overall Join Strategies

Figure 10 shows a comparison of DSM Post-Projection using Radix-Decluster with NSM Pre-Projection, DSM Pre-Projection, and two NSM Post-Projection variants: our own Radix-Decluster and Jive-Join [LR99]. All these variants use the cache-conscious Partitioned Hash-Join; they vary only in the projection strategy. To show the overall effect of all cache optimizations, we also include NSM Pre-Projection with naive non-partitioned Hash-Join ("NSM-pre-hash").

To analyze the impact of all parameters (π, N, h), Figure 10 depicts three plots, each varying one parameter while keeping the others fixed. We observed similar behavior in experiments with different values for the fixed parameters.

Figure 10b shows that with decreased hit-rate, all strategies become cheaper (due to the smaller join result) but DSM Post-Projection even more, which is explained by the decreased overall impact of the projection phase (with the relatively expensive Radix-Decluster), with respect to the cost for creating the join-index with Partitioned Hash-Join.

Figure 10c shows that all strategies scale linearly with cardinality. The steeper increase of DSM Post-Projection ("DSM-post-decluster") in the lower range of π occurs because on small cardinalities, individual columns fit in the cache, such that the relatively expensive Radix-Decluster is not necessary, as indicated by the point types that identify the projection method used for both the left and the right table (with the one-letter codes defined in Section 4.1).

Pre-Projection Alternatives Most systems other than MonetDB that use DSM or other forms of vertical fragmentation, such as transposed files [Bat79] or projection indices [OQ97], use a scan operator that scans all columns simultaneously (called `Assemble()` in [RDS02]).

One factor to consider in all our comparisons is that DSM Post-Projection has a CPU efficiency advantage over all other alternatives. Due to the column-at-a-time execution in MonetDB, its operators have "zero degree of freedom", such that in their implementation a hard-coded operation on a hard-coded type is executed in a tight inner loop that iterates over large arrays. Modern compilers can handle such code well, achieving high IPC by e.g., loop pipelining. The other strategies handle all projection columns simultaneously (tuple-at-a-time), and have to deal with some degree of freedom, namely a list of projection columns, which is passed at run-time (additionally, the NSM strategies have to extract column values from a NSM record by looking at record offsets stored in a table header). Such code not only has to perform some more work (CPU overhead) but the additional complexity and dependencies in the inner loops are bound to hinder the compiler in getting a good IPC.

The main difference in Figure 10a between DSM Pre-Projection ("DSM-pre-phash") and DSM Post-Projection is this very CPU advantage of the latter. A second smaller difference is that as Pre-Projection handles all projections at the same time (during the join), less tuples fit in the clusters created by Radix-Cluster, such that it more quickly needs multiple passes. This is again compounded by the CPU disadvantage, allowing it to trade less extra CPU for better memory access (e.g., two-pass Radix-Cluster for creating many clusters almost never wins, leaving the strategy with a bad memory access pattern).

The difference between DSM Pre-Projection and NSM Pre-Projection ("NSM-pre-phash") is mainly in the better cache-line usage of DSM. On the positive side, the projections done by the Radix-Clustering of the NSM relations access the input relation sequentially. Thus, even if cache-lines are used sparsely, the pain will be reduced somewhat by automatic memory prefetching on modern hardware (it is "only" a bandwidth problem). As can be seen in Figure 10a, this impact is only considerable at low π.

Finally, the big difference in NSM Pre-Projection between non-partitioned and Partitioned Hash-Join is explained by the performance hit taken by uncachable random memory access. As the projectivity π increases, naive Hash-Join uses its cache lines relatively better, and it approaches Partitioned Hash-Join (but on no occasion surpasses it).

NSM Post-Projection Alternatives The performance of NSM Pre-Projection at $\pi = 1$ in Figure 10a roughly corresponds to the first phase (the creation of the join-index) in the NSM Post-Projection strategies. This cost is considerable, giving both Radix-Decluster on NSM ("NSM-post-decluster") and Jive-Join ("NSM-post-jive") a hard time competing with the other strategies, as creating the join-index is only their first step. Subsequently, they need to access the wide NSM base tables one more time for performing the projections. This would of course have been very different had we assumed the (clustered) join-index to be already present as an accelerator structure. As we concentrate on large ad-hoc joins, however, the join-index cannot have been precomputed.

Jive-Join first sorts the join-index, and then carries out a special Positional-Join ("Left Jive-Join") with the one join input, that directly re-sorts its output on the `oids` of the other table. It generates two separate outputs, in the same order (which is the final result order), one containing the clustered `oids`, the other containing all projection columns from the first join input. In the second phase, a second special Positional-Join ("Right Jive-Join") is done between each cluster of `oids` (that is first sorted for better access) and the second table, where the results are written back in the order of the result (the order of the `oids` before re-sorting) [LR99].

As the detailed performance results on Left and Right Jive-Join in Figures 9e and 9f show, the Left Jive-Join phase may suffer from a too high cluster fanout in much the same way single-pass Radix-Cluster does, while the Right Jive-Join may suffer from too few (=big) clusters, much like Partitioned Hash-Join does. However, the strategy of creating not too many cluster in the first phase, then refining them with Radix-Cluster in order not to have too big clusters in the Right Jive-Join, does not work as then the reordering in Right Jive-Join has random access to a too large cluster.

The scalability of both Radix-Decluster as well as Jive-Join is limited to $O(C^2/T^2)$, where T is the tuple width. Therefore, on large cardinalities, wide NSM tuples can quickly get these algorithms into cache problems,

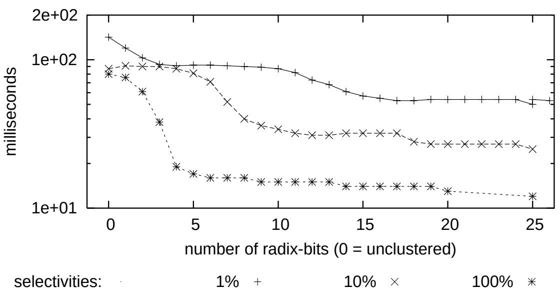


Figure 11: Impact of Selectivity: Sparse Clustered Positional Join ($N = 1M$)

limiting their applicability for cache-conscious join.

Sparse Projections Sparse projections occur when a join relation is a selection on a base table. Figure 11 shows that the performance of Positional-Join suffers significantly with a decreasing selection percentage. This is more of an issue for DSM than for NSM, as in DSM cache-lines hold values of multiple consecutive tuples, and if only a small percentage is used, sequential RAM bandwidth utilization decreases. In NSM, cache-lines typically hold only values of a single tuple, and bandwidth efficiency mainly depends on *projectivity*, not on selectivity. Still, this need not be a show-stopper, as sequential RAM bandwidth is in rather generous supply and unlike latency shows steady progress as hardware evolves.

The effect of sparse projections on DSM Post-Projection is also shown in all Figures 10a,b,c using error bars. The smallest error bar shows performance with 10% selectivity (i.e., cardinality of the underlying base-table is $10N$) and the second corresponds to 1% selectivity (cardinality is $100N$). While we see that DSM Post-Projection performance decreases with a lower selectivity percentage, it clearly stays the better strategy overall.

We should note that this comparison is worst-case for DSM Post-Projection. First, for brevity we omitted the sparse access data for NSM, which is also affected by sparse access, only to a much lesser degree. Second, if the selectivity is low, such as 1% or less, then in many cases the intermediate relation would become small, making the join an “easy” instead of a “hard” case (see Section 3). For “easy” joins, DSM Post-Projection could use an u/u strategy, thus significantly improving its performance.

5. DSM Radix-Decluster in a NSM DBMS

Our results strongly suggest that RDBMS performance can be enhanced by introducing vertical fragmentation as an accelerator structure, i.e., projection indices [OQ97]. Such a “DSM-subsystem” would profit in OLAP queries that touch many tuples but few columns, and would preferably use CPU-efficient MonetDB-like hard-coded operators that manipulate columns at-a-time, such as Positional-Join, Radix-Cluster and Radix-Decluster. The very purpose of MonetDB’s cache-conscious query processing algorithms is to restrict all random access to very small ranges that fit the CPU caches. Thus, the only I/O access to the DSM fragments are sequential bulk reads and writes. On our evaluation platform, our algorithms caused read and write rates between 200MB/s and 500MB/s, which can be supported using a PCI-X RAID consisting of 12 SCSI disks.[5]

A case for a DBMS with mixed DSM-NSM storage is made in [RDS02], which also describes how updates could be accommodated efficiently using differential files to the DSM file images. In such an architecture, a buffer manager would still be used as an efficient means of well-controlled (asynchronous) I/O. In MonetDB, however, columns are contiguous arrays, while in an RDBMS the columns would be stored in pages at various locations of the buffer pool. Therefore, the Radix-Decluster technique of inserting “by position” in the insertion window would not apply directly. Finding the correct page and offset would be especially difficult if we were to

[5]Preliminary experiments with lightweight data (de-)compression indicate that a negligible CPU investment can more than half the needed I/O bandwidth on problems like TPC-H. As I/O bandwidth is precious, this looks a worthwhile approach to help scale DSM to disk-based scenarios.

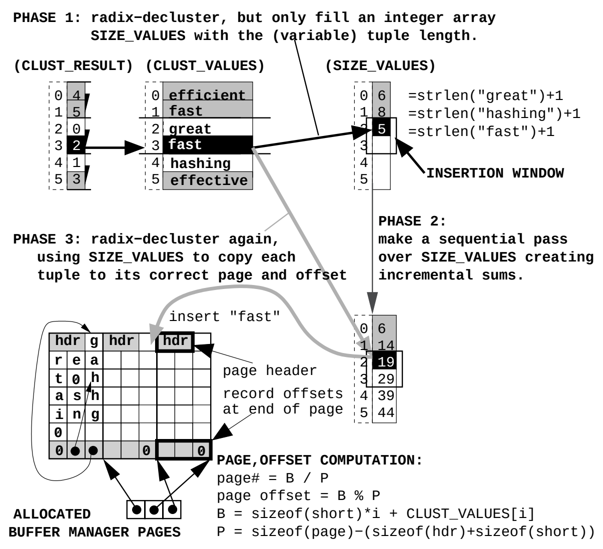

PHASE 1: radix-decluster, but only fill an integer array SIZE_VALUES with the (variable) tuple length.
(CLUST_RESULT)
(CLUST_VALUES)
(SIZE_VALUES)
efficient
fast
great
fast
hashing
effective
=strlen("great")+1
=strlen("hashing")+1
=strlen("fast")+1
INSERTION WINDOW
PHASE 2: make a sequential pass over SIZE_VALUES creating incremental sums.
PHASE 3: radix-decluster again, using SIZE_VALUES to copy each tuple to its correct page and offset
insert "fast"
hdr
page header
record offsets at end of page
PAGE,OFFSET COMPUTATION:
page# = B / P
page offset = B % P
B = sizeof(short)*i + CLUST_VALUES[i]
P = sizeof(page)−(sizeof(hdr)+sizeof(short))
ALLOCATED BUFFER MANAGER PAGES

Figure 12: Handling Non-Continuous Addressing and Variable-sized Data

handle variable-sized values such as strings. Figure 12 shows how both problems are solved in a buffer manager that uses NSM-like pages for storing sequences of variable-size values. Output space has been allocated in a number of buffer pages, whose start addresses are stored in an index array. First, the Radix-Decluster is executed, but it does not insert any values, but just records the lengths of the variable-size values in an extra integer array. This temporary array is, of course, addressable by position. In a second phase, the lengths are summed to calculate locations. In a third phase, the Radix-Decluster operation is re-executed to copy values into the result, and this time the correct page and offset for each value can be calculated, using the computed location accessible by position in the array. Note that for fixed-size values, the extra passes are not even necessary, and page and offset can be determined from the `oid`, which is the result tuple sequence number.

6. Conclusion

We have investigated the problem of performing large equi-joins with projections in a cache-conscious manner. As can be seen in the left graph of Figure 10, performance may vary by more than an order of magnitude with different relation projectivity, thus proving that projection cost can have a strong impact on overall join efficiency.

Our main contribution, the Radix-Decluster algorithm, is the crucial tool of MonetDB to process (i.e., join, but also re-order) huge tables with a good access pattern, both in terms of CPU cache access as well as I/O access (through virtual memory).

In our experiments, we tested various cache-conscious join (projection) strategies both on the NSM and DSM storage schemes. One important conclusion from these experiments is that Partitioned Hash-Join significantly improves performance not only for MonetDB and DSM, but also for the NSM pre-projection strategy, as is used by all standard RDBMS products (compare in Figure 10 the non-cache-friendly “NSM pre-hash” with “NSM pre-phash”), proving that this algorithm carries generic merit.

The performance evaluation further shows that Radix-Decluster is pivotal in making DSM post-projection the most efficient overall strategy. We should note, that unlike Radix-Cluster, Radix-Decluster is a single-pass algorithm, and thus has a scalability limit imposed by a maximum number of clusters and thus tuples. This limit depends on the CPU cache size and is quite generous (assuming four-byte column values, the 512KB cache of a Pentium4 Xeon allows to project relations of up to half a billion tuples) and scales quadratically with the cache size (so the 6MB Itanium2 cache allows for 72 billion tuples).

This limitation also explains why Radix-Decluster is less successful in NSM post-projection, as its scalability is also inversely quadratically related to the tuple width. Rephrased positively, vertical fragmentation (DSM) and column-wise execution reduce tuple width, fit more tuples in the CPU cache and quadratically improve scalability. For NSM, however, we find the “traditional” pre-projection technique to work best, also outperforming the alternative NSM post-projection strategy of Jive-Join, which was not intended as a generic join method, but rather for exploiting precomputed join-indices.

As for the prospects of applying DSM Radix-Decluster in off-the-shelf RDBMS products, we support the case made in [RDS02] for systems that combine DSM and NSM natively, or that simply add DSM to the normal NSM representation as *projection indices* [OQ97], and show how such disk-based systems could use our Radix-Algorithms through their buffer manager.

References

[ADHS01] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proc. VLDB Conf.*, pages 169–180, Roma, Italy, September 2001.

[ADHW99] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood. DBMSs on modern processors: Where does time go? In *Proc. VLDB Conf.*, pages 266–277, Edinburgh, Scotland, UK, September 1999.

[Bat79] D. Batory. On Searching Transposed Files. *TODS*, 4(4):531–544, 1979.

[BGB98] L. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization Of Commercial Workloads. In *Proc. ISCA*, Barcelona, Spain, June 1998.

[BMK99] P. Boncz, S. Manegold, and M. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. VLDB Conf.*, pages 54–65, Edinburgh, Scotland, UK, September 1999.

[Bon02] P. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, UVA, Amsterdam, The Netherlands, May 2002.

[CK85] G. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD Conf.*, pages 268–279, Austin, TX, USA, May 1985.

[HSU$^+$01] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. In *Intel Technology Journal*, http:// developer.intel.com/ technology/ itj/, February 2001.

[KPH$^+$98] K. Keeton, D. Patterson, Y. He, A. Raphael, and W. Baker. Performance Characterization of a quad Pentium Pro SMP using OLTP workloads. In *Proc. ISCA*, pages 15–26, Barcelona, Spain, June 1998.

[LH99] G. Lauterbach and T. Horel. UltraSparc-III: designing 3rd generation 64bit platforms. *IEEE Micro*, 19(3):56–66, 1999.

[LR99] Z. Li and K. Ross. Fast Joins Using Join Indices. *The VLDB Journal*, 8(1):1–24, 1999.

[Man02] S. Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance*. PhD thesis, UVA, Amsterdam, The Netherlands, December 2002.

[MBK02] S. Manegold, P. Boncz, and M. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *Proc. VLDB Conf.*, pages 191–202, Hong Kong, China, August 2002.

[McC95] A. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Technical Committee on Computer Architecture newsletter*, December 1995.

[OQ97] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *Proc. SIGMOD Conf.*, pages 38–49, Tucson, AZ, USA, May 1997.

[RDS02] R. Ramamurthy, D. DeWitt, and Q. Su. A Case for Fractured Mirrors. In *Proc. VLDB Conf.*, pages 430–441, Hong Kong, China, August 2002.

[SKN94] A. Shatdahl, C. Kant, and J. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proc. VLDB Conf.*, pages 510–512, Santiago, Chile, September 1994.

[Syb96] Sybase Corp. Whitepaper. *Adaptive Server IQ*, July 1996. http:// www. sybase. com/ content/ 1008840/ iq_wp_l00899.pdf.

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

A. Cost Models

The techniques proposed by Manegold ([MBK02, Man02]) allow us to define cost models for database algorithms by simply describing their data access patterns. For details on how to derive the actual cost functions from the access pattern descriptions, we refer the interested reader to [MBK02, Man02].

We now shortly recap the access patterns for Radix-Cluster and Partitioned Hash-Join, before we point our attention to the variants of Positional Join and Radix-Decluster as proposed in this paper. To compare our Radix-Decluster algorithm with Jive-Join [LR99], we also discuss the access patterns of Jive-Join.

Radix-Cluster During each pass, Radix-Cluster scans the entire input sequentially ("s_trav"). On the output side, the $H_p = 2^{B_p}$ clusters are access in a random ("ran") order, but within each cluster, access is sequential. Input and output are accessed concurrently, hence we get:

$$\{X_j\}|_{j=1}^{2^B} \leftarrow \mathit{radix\_cluster}(X,B,P):$$

$$\oplus|_{p=1}^{P}\left(\mathsf{s\_trav}(X) \odot \mathsf{nest}\left(\{X_j\}|_{j=1}^{2^{B_p}}, 2^{B_p}, \mathsf{s\_trav}(X_j), \mathtt{ran}\right)\right)$$

Partitioned Hash-Join Hash-Join consists of two phases. During the build phase, the entire inner input is sequentially read while the entire hash table is accessed in random order ("r_trav") when inserting the keys. During the probe phase, the entire outer input is sequentially read, each key is looked-up in the hash table (random access, "r_acc"), and the output is again written in sequential order. Partitioned Hash-Join perform a simple Hash-Join for each the 2^B pairs of matching clusters, hence:

symbol	description
R,X,Y,Z	data regions (cf., [MBK02, Man02])
$N = \lvert R\rvert$	length of data region R [number of data items]
$\overline{\underline{R}}$	width of data region R (size of data items) [bytes]
$\lvert\lvert R\rvert\rvert = \lvert R\rvert \cdot \overline{\underline{R}}$	size of data region R [in bytes]
$\{R_j\}\|_{j=1}^{n}$	data region R clustered in n partitions
W	radix-decluster insertion window (a data region)
$w = \lvert W\rvert/2^B$	tuples read per cluster per insertion window
$\#w = \lvert X'\rvert/\lvert W\rvert$	number of insertion windows
C	cache capacity [in bytes]
ω	number of attributes in a (base) table
π	number of projection attributes in a query
h	join hit rate
P	number of radix-cluster passes
B, B_p	number of radix-bits (total and per pass)
H, H_p	number of clusters or partitions (total, per pass)
nest	interleaved multi-cursor access
r_acc	random access
s_trav, rs_trav	single and repetitive sequential traversal
r_trav, rr_trav	single and repetitive random traversal
$\oplus, \odot$	sequential and concurrent execution
$\circledcirc\|_{q=1}^{p}(P_q)$	$\equiv \circledcirc(P_1,\ldots,P_p) \equiv P_1 \circledcirc \ldots \circledcirc P_p \quad (\circledcirc \in \{\oplus,\odot\})$

Table 1: List of Symbols used in Cost Models (cf., [Man02])

$Y' \leftarrow hash\_build(Y):$

$$\mathsf{s\_trav}(Y) \odot \mathsf{r\_trav}(Y') \qquad =: \mathsf{build\_hash}(Y,Y')$$

$Z \leftarrow hash\_probe(X,Y'):$

$$\mathsf{s\_trav}(X) \odot \mathsf{r\_acc}(|X|,Y') \odot \mathsf{s\_trav}(Z) \qquad =: \mathsf{probe\_hash}(X,Y',Z)$$

$Z \leftarrow hash\_join(X,Y):$

$$\mathsf{build\_hash}(Y,Y') \oplus \mathsf{probe\_hash}(X,Y',Z) \qquad =: \mathsf{hash\_join}(X,Y,Z)$$

$\{Z_p\}|_{p=1}^{2^B} \leftarrow part\_hash\_join(\{X_p\}|_{p=1}^{2^B},\{Y_p\}|_{p=1}^{2^B},B):$

$$\oplus|_{p=1}^{2^B}(\mathsf{hash\_join}(X_p,Y_p,Z_p))$$

Positional-Join Like the probe phase of Hash-Join, all versions of Positional-Join perform sequential traversals of both the outer input and the output. In case the (left) input join attribute is already ordered lookups to the inner (right) input make up a sequential traversal; otherwise, the lookup comprise a random access on the inner input. The Clustered version processes the clusters one after another, performing Unsorted Positional-Join on each pair of matching clusters. Hence:

$Z \leftarrow unsort\_pos\_join(X,Y):$

$$\mathsf{s\_trav}(X) \odot \mathsf{r\_acc}(|X|,Y) \odot \mathsf{s\_trav}(Z) \qquad =: \mathsf{unsort\_pos\_join}(X,Y,Z)$$

$Z \leftarrow sort\_pos\_join(X,Y):$

$$\mathsf{s\_trav}(X) \odot \mathsf{s\_trav}(Y) \odot \mathsf{s\_trav}(Z) \qquad =: \mathsf{sort\_pos\_join}(X,Y,Z)$$

$\{Z_p\}|_{p=1}^{2^B} \leftarrow clust\_pos\_join(\{X_p\}|_{p=1}^{2^B},\{Y_p\}|_{p=1}^{2^B},B):$

$$\oplus|_{p=1}^{2^B}(\mathsf{unsort\_pos\_join}(X_p,Y_p,Z_p))$$

Radix-Decluster For each of the $\#w$ insertion windows of the final output, Radix-Decluster sequentially traverses the entire CLUST_BORDERS. During each of the $\#w$ iterations, on average $(1/\#w)$-th of each of the 2^B clusters of CLUST_VALUES and CLUST_RESULT are read sequentially. For each cluster, the tuples that fall into the current insertion window are written to their final position in the output. Effectively, this means, that the insertion window X'_k is 2^B times traversed in random order, where each traversal touches only $(1/2^B)$-th of X'_k; in other word, each traversal uses an average access stride of $2^B * \underline{X}$. In total, we get:

CLUST_VALUES: $\{X_j\}|_{j=1}^{2^B} \quad \langle |X| = N, \underline{X} = 4\rangle$

CLUST_RESULT: $\{Y_j\}|_{j=1}^{2^B} \quad \langle |Y| = N, \underline{Y} = 4\rangle$

CLUST_BORDERS: $Z \quad \langle |Z| = 2^B, \underline{Z} = 4\rangle$

$\{X'_k\}|_{k=1}^{\#w} \leftarrow radix\_decluster(\{X_j\}|_{j=1}^{2^B},\{Y_j\}|_{j=1}^{2^B},Z,\#w):$

$$\oplus|_{k=1}^{\#w}\left(\oplus|_{j=1}^{2^B}\left(\mathsf{s\_trav}(\frac{X_j}{\#w}) \odot \mathsf{s\_trav}(\frac{Y_j}{\#w})\right) \odot \mathsf{rr\_trav}(2^B,X'_k,2^B * \underline{X})\right) \odot \mathsf{rs\_trav}(\#w,Z)$$

Jive-Join With the pre-computed join-index being sorted in order of the RowIds of the left projection table, the first phase of Jive-Join performs a merge join between the join-index and the left table, scanning both sequentially. The output of the first phase consists of the left half of the final result and a re-ordered join-index. Both parts are clustered such that each cluster of the re-ordered join-index contain a consecutive range of the

RowIds of the right projection table. Hence, the access patterns on both outputs is similar to that of Radix-Cluster:

$$
\begin{aligned}
\left\langle \{Z_p\}|_{p=1}^{2^B}, \{Y'_p\}|_{p=1}^{2^B} \right\rangle \leftarrow \mathit{left\_jive\_join}(X,Y,B) : & \\
\mathsf{s\_trav}(X) \odot\ & \mathsf{s\_trav}(Y) \\
\odot\ & \mathsf{nest}\left(\{Z_p\}|_{p=1}^{2^B}, 2^B, \mathsf{s\_trav}(Z_p), \mathtt{ran}\right) \\
\odot\ & \mathsf{nest}\left(\{Y'_p\}|_{p=1}^{2^B}, 2^B, \mathsf{s\_trav}(Y'_p), \mathtt{ran}\right)
\end{aligned}
$$

For each cluster of the re-ordered join-index, the second phase of Jive-Join then performs a merge-join with the respective cluster of the left table, traversing both sequentially. To match the order of the left half of the output, the right half of the output then has to be written in random order:

$$
\begin{aligned}
\{Z_p\}|_{p=1}^{2^B} \leftarrow \mathit{right\_jive\_join}(X, \{Y_p\}|_{p=1}^{2^B}, B) : & \\
\oplus|_{p=1}^{2^B} \left(\mathsf{s\_trav}(X_p) \odot \mathsf{s\_trav}(Y_p) \odot \mathsf{r\_trav}(Z_p)\right) &
\end{aligned}
$$