Monet; a next-Generation DBMS Kernel For Query-Intensive Applications

Boncz, P.A.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 6

Memory/CPU Optimized Query Processing

Custom hardware—from workstations to PCs—has experienced tremendous improvements in the past decades. However, as discussed in Section 3.2.2, this improvement has not been equally distributed over all aspects of hardware performance and capacity. Figure 3.1 shows that the speed of commercial microprocessors has been increasing roughly 70% every year, while the speed of commodity DRAM has improved by little more than 50% over the past decade [Mow94]. A partial reason for this trend is that there is a direct trade-off between capacity and speed in DRAM chips, and the highest priority has been for increasing capacity. The result is that from the perspective of the processor, memory has been getting slower at a dramatic rate. This affects all computer systems, making it increasingly difficult to achieve high processor efficiency.

Another trend is the ever increasing number inter-stage and intra-stage parallel execution opportunities provided by the multiple execution pipelines and speculative execution in modern CPUs. It has been shown that current database systems on the market make poor use of these new features [ADHW99, KPH+98, BGB98, TLPZT97]. This poor use is embodied by low IPC scores (instructions executed per CPU cycle) and high number of memory cache misses observed during database query execution. In contrast, other computational fields, like scientific computation, show over the generations of hardware increasing IPC scores, achieving near optimum performance out of modern CPUs and memory subsystems.

One rationale behind Monet was to reverse this trend in database systems; therefore, its design focuses on efficient use of main memory and CPU resources in query intensive database applications. One research question in Monet therefore was how data structures and algorithms should be designed in order to make efficient use of these resources. This chapter treats this subject, in particular the efficient processing of large joins in the context of (standard) relational query processing.

6.1 Related Work

Database system research into the design of algorithms and data structures that optimize memory access, has been relatively scarce. Our major reference here is the work by ShatDahl et al. [SKN94], which shows that join performance can be improved using
a main memory variant of Grace Join, in which both relations are first hash-partitioned in chunks that fit the (L2) memory cache. While being a landmark paper, there were various reasons that led us to explore this direction of research further. First, after its publication, the observed trends in custom hardware have continued, deepening the memory access bottleneck. For instance, the authors list a mean performance penalty for a cache miss of 20-30 cycles in 1994, while a range of 200-300 is typical in 2001 (and rising). This increases the benefits of cache-optimizations, and possibly changes the trade-offs. Another development has been the introduction of so-called level-one (L1) caches, which are typically very small regions on the CPU chip that can be accessed at almost CPU clock-speed. The authors of [SKN94] provide algorithms that are only feasible for the relatively larger, off-chip L2 caches that were in existence at that time. Finally, [SKN94] uses standard relational data structures. We will argue, in contrast, that the impact of memory access is so severe, that vertically fragmented data structures should be applied at the physical level of database storage.

Though we believe in the universal relevance of memory-access optimization to database performance, it is especially important for main memory databasees [LC86a, LC86b, Eic89, Wil91, AP92, GMS92]. In the case of Monet, we use aggressive coding techniques for optimizing CPU resource utilization [BK99], that go much beyond the usual MMDBMS implementation techniques [DKO+84]. As described in Section 5.3.3, all Monet implementation code is written in a macro language, from which C language implementations are generated. The macros implement a variety of techniques, by virtue of which the inner loops of performance-critical algorithms like join are free of overheads like database ADT calls, data movement and loop condition management. These techniques were either pioneered by our group (e.g., logarithmic code expansion [Ker89]) or taken from the field of high performance computing [LL97]. In this work, we will show that these techniques allow compilers to produce code that better exploits the parallel resources offered by modern CPUs.

Past work on main-memory query optimization [LN96, WK90] models the main-memory cost of query processing operators on the coarse level of a procedure call, using profiling to obtain some 'magical' constants. As such, these models do not provide insight in individual components that make up query cost, limiting their predictive value. Conventional (i.e., non main-memory) cost modeling, in contrast, has I/O as dominant cost aspect, making it possible to formulate accurate models based on the amount of predicted I/O work. Calibrating such models is easy, as statistics on the I/O accesses caused during an experiment are readily available in a database system. Past main memory work has not been able to provide such cost models on a similarly detailed level, for two reasons. First, it has been found difficult to model the interaction between low-level hardware components like CPU, Memory Management Unit, bus and memory caches. Second, it was impossible to measure the status of these components during experiments, which is necessary for tuning and calibration of models. Modern CPUs, however, contain counters for events like cache misses, and exact CPU cycles [BZ98, ZLT96, Yea96, Adv00, GBC+95, GT96, LH99, HSU+01, SA00, TDF01]. This enabled us to develop a new main-memory cost modeling methodology that first mimics the memory access pattern of an algorithm, yielding a number of CPU cycle and memory cache events, and then scores this pattern with an exact cost prediction. Therefore, the main contribution of the algorithms, models and experiments presented here is to demonstrate that detailed cost-modeling of main memory performance is both important and feasible.
6.2 Outline

In Section 6.3, we take a look at the aspects of modern computer architecture most relevant for the performance of main memory query execution, identify ongoing trends in custom hardware, and outline the consequences of these trends for database architecture. We also describe our calibration tool, that extracts the most important hardware characteristics like cache size, cache line-size, and cache latency from any computer system, and provide results for our benchmark platforms (Sun, SGI, and PC hardware).

In Section 6.4.2, we introduce the Radix-Cluster algorithm that improves the partitioning phase in Partitioned Hash-Join, as it allows to trade memory access cost for extra CPU processing. We perform exhaustive experiments where we use CPU event counters to obtain detailed insight in the performance of this algorithm. First, we vary the partition sizes, to show the effect of tuning the memory access pattern to the memory caches sizes. Second, we investigate the impact of code optimization techniques from the field of main memory databases. These experiments show that improvements of almost an order of magnitude can be obtained by combining both techniques (cache tuning and code optimization) rather than by each one individually. We explain our results with detailed models of both the partition (Radix-Cluster) and join phase of Partitioned Hash-Join, and show how performance can exactly be predicted from hardware events like cache and TLB misses.

The partitioned join and clustering strategies solely focus on determining for an equip-join which tuples join together. Their end-result hence is a join-index; i.e. projection of columns other than the join key columns from both input tables into the result was disregarded. In traditional relational DBMSs, this is not a critical omission, as including projection columns in the join algorithms is a trivial extension on the Radix-Cluster and Partitioned Hash-Join (the relations processed just get a bit wider due to the extra “luggage” of the projection columns). This is not the case for Monet query processing with vertically fragmented relations, where column projection is a separate phase, posterior to computing the join itself. In Section 6.5, we describe a new algorithm called Radix-Decluster, that executes such single-column projections very efficiently with regards to CPU usage and memory access, and use it in extensive benchmarking to compare the performance of overall join processing (including projections) of Monet with “traditional” and cache-conscious join processing in relational systems.

6.3 Modern Hardware and DBMS Performance

First, we describe the technical details of modern hardware relevant for main memory query performance, focusing on CPU and memory architectures. We perform experiments to illustrate how the balance between CPU and memory cost in query processing has shifted through time, and discuss a calibration tool that automatically extracts the hardware parameters most important for performance prediction from any computer system. We then look at what future hardware technology has in store, and identify a number of trends.
6.3.1 A Short Hardware Primer

While CPU clock frequency has been following Moore’s law (doubling every three years), CPUs have additionally become faster through parallelism within the processor. Scalar CPUs separate different execution stages for instructions, e.g., allowing a computation stage of one instruction to be overlapped with the decoding stage of the next instruction. Such a pipelined design allows for inter-stage parallelism. Modern super-scalar CPUs add intra-stage parallelism, as they have multiple copies of certain (pipelined) units that can be active simultaneously. Although CPUs are commonly classified as either RISC or CISC, modern CPUs combine successful features of both. Figure 6.1 shows a simplified schema that characterizes how modern CPUs work: instructions that need to be executed are loaded from memory by a fetch-and-decode unit. In order to speed up this process, multiple fetch-and-decode units may be present (e.g., the PentiumIII has three, the R10000 two [Die99, KP99, ZLT96, Yea96]). Decoded instructions are placed in an instruction queue, from which they are executed by one of various functional units, which are sometimes specialized in integer-, floating-point, and load/store pipelines. The PentiumIII, for instance, has two such functional units, whereas the R10000 has even five. To exploit this parallel potential, modern CPUs rely on techniques like branch prediction to predict which instruction will be next before the previous has finished. Also, the modern cache memories are non-blocking, which means that a cache miss does not stall the CPU. Such a design allows the pipelines to be filled with multiple instructions that will probably have to be executed (a.k.a. speculative execution), betting on yet unknown outcomes of previous instructions. All this goes accompanied by the necessary logic to restore order in case of mis-predicted branches. As this can cost a significant penalty, and as it is very important to fill all pipelines to obtain the performance potential of the CPU, much attention is paid in hardware design to efficient branch prediction. CPUs work with prediction tables that record statistics about branches taken in the past.

Modern computer architectures have a hierarchical memory system, as depicted in Figure 6.2, where access by the CPU to main memory, consisting of DRAM chips on the system board, is accelerated by various levels of cache memories. Introduction of these cache memories, that consist of fast but expensive SRAM chips, was necessary due to the fact that DRAM memory latency has progressed little through time, making
its performance relative to the CPU become worse exponentially. First, one level of cache was added by placing SRAM chips on the motherboard. Then, as CPU clock-speeds kept increasing, the physical distance between these chips and the CPU became a problem, as it takes a minimum amount of time per distance to carry an electrical signal over a wire. As a result, modern CPUs have cache memories inside the processor chip. Without loss of generality, we assume one on-chip cache called L1, and a typically larger off-chip cache on the system board called L2. We identify three aspects that determine memory access cost:

**latency** Our exact definition of memory latency \( (l_{mem}) \) is the time needed to transfer one byte from the main memory to the L2 cache. This occurs, when the piece of memory being accessed is in neither the L1 nor the L2 cache, so we speak of an \( L2 \) miss. It is important to note that during this time, all current hardware actually transfers multiple consecutive words to the memory subsystem, since each cache level has a smallest unit of transfer (called the cache line). During one memory fetch, modern hardware loads an entire cache line from the main memory\(^1\) in one go, by reading data from many DRAM chips at the same time, transferring all bits in the cache line in parallel over a wide bus. Similarly, with \( L2 \) latency \( (l_{l,2}) \) we mean the time it takes the CPU to access data that is in L2 but not in L1 (an \( L1 \) miss), and \( L1 \) latency \( (l_{l,1}) \) is the time it takes the CPU to access data in L1. Each L2 miss is preceded by an L1 miss. Hence, the total latency to access data that is in neither cache is \( l_{mem} + l_{L2} + l_{L1} \). As L1 latency cannot be avoided, we assume in the remainder of this chapter, that L1 latency is included in the pure CPU cost, and regard only memory latency and \( L2 \) latency as explicit memory access cost.

To give an idea of the typical latencies, the Origin2000 used in our experiments has a L1 latency of 1 cycle, a L2 latency of 6 cycles and a memory latency of 100 cycles (it uses the 250MHz R10000 processors [ZLTi96, Yea96], so 1 cycle = 4ns). Thus, loading an L1 line (32 byte) form memory to the CPU takes \( l_{mem} + l_{L2} = 106 \) cycles, loading an L2 line (128 byte, 4 L1 lines) from memory to CPU takes \( l_{mem} + 4 * l_{L2} = 124 \) cycles, or 496ns.

**bandwidth** We define memory bandwidth as the number of megabytes of main memory the CPU can access per second. Sometimes there is a difference between read and write bandwidth, but this difference tends to be small. Bandwidth is usually maximized on a sequential access pattern, as only then all memory words in the cache lines are used fully. In conventional hardware, the memory bandwidth used to be simply the cache line size divided by the memory latency, but modern multiprocessor systems typically provide excess bandwidth capacity.

For instance, when fetching cache lines sequentially our Origin2000 would provide \( 128 / (496 \times 10^{-9}) = 246 \)Mb/s, but its maximum bandwidth is in fact 555MB/s.

**address translation** The Translation Lookaside Buffer (TLB) is a common element in modern CPUs (see Figure 6.1). This buffer is used in the translation of logical virtual memory addresses used by application code to physical page addresses in

\(^1\)To which cache line the memory is loaded, is determined from the memory address. An X-way associative cache allows to load a line in X different positions. If \( X > 1 \), some cache replacement policy chooses one from the X candidates. Least Recently Used (LRU) is the most common replacement algorithm.
CHAPTER 6. MEMORY/CPU OPTIMIZED QUERY PROCESSING

Figure 6.3: CPU and memory access cost per tuple in a simple table scan

the main memory of the computer. The TLB is a kind of cache that holds the translation for (typically) the 64 most recently used pages. If a logical address is found in the TLB, the translation has no additional cost. However, if a logical address is not cached in the TLB, a TLB miss occurs. A TLB miss is handled by trapping to a routine in the operating system kernel, that translates the address and places it in the TLB. Depending on the implementation and hardware architecture, TLB misses can be more costly even than a main memory access (on our Origin2000, it costs 57 cycles = 228ns). Moreover, handling a TLB miss often involves a lookup in a large memory array, whose access can itself trigger additional memory cache misses. The more pages an application uses (which is also dependent of the often configurable size of the memory pages), the higher the probability of TLB misses.

6.3.2 Experimental Quantification

We use a simple scan test to demonstrate the severe impact of memory access cost on the performance of elementary database operations. In this test, we sequentially scan an in-memory buffer, by iteratively reading one byte with a varying stride, i.e. the offset between two subsequently accessed memory addresses. We made sure that the buffer was in memory, but not in any of the memory caches. This experiment mimics what happens if a database server performs a read-only scan of a one-byte column in an in-memory table with a certain record-width (the stride); as would happen in a in a simple aggregation (e.g. SELECT MAX(column) FROM table).

Figure 6.3 shows results of this experiment on a number of popular workstations of the past decade. The X-axis shows the different systems ordered by their age, and per system the different strides tested. The Y-axis shows the absolute elapsed time for the experiments. For each system, the graph is split up to show which part of the elapsed time is spent waiting for memory (upper), and which part with CPU processing (lower, gray-shaded).

All systems show the same basic behavior with best performance at stride 1, increasing to some maximum at a larger stride, after which performance stays constant. This
is explained as follows: when the stride is small, successive iterations in the scan read bytes that are near to each other in memory, hitting the same cache line. The number of L1 and L2 cache misses is therefore low, and the memory access costs are negligible compared to the CPU costs. As the stride increases, the cache miss rates and thus the memory access costs also increase. The cache miss rates reach their maxima, as soon as the stride reaches the cache line size. Then, every memory read is a cache miss. Performance cannot become any worse and stays constant.

When comparing the Sun LX to the Origin2000, we see that CPU performance has increased 10-fold, of which a factor 5 can be attributed to faster clock frequency (from 50MHz to 250MHz), and a factor 2 to increased processor parallelism (the CPU cost has fallen from 160ns at 50MHz = 8 cycles to 16ns at 250MHz = 4 cycles). While this trend of exponentially increasing CPU performance is easily recognizable, the memory cost trend in Figure 6.3 shows a mixed picture, and has certainly not kept up with the advances in CPU power. Consequently, while our experiment was still largely CPU-bound on the Sun from 1992, it is dominated by memory access cost on the modern machines (even the PentiumIII with fast memory is 75\% of the time stalling for memory). Note that the modern machines from Sun, Silicon Graphics and DEC actually have a memory access cost that in absolute numbers is even higher than the Sun from 1992. This can be attributed to the complex memory subsystem that goes with multi-processor SMP design, resulting in a high memory latency. These machines do provide a high memory bandwidth—thanks to the ever growing cache line sizes\(^2\)—but this does not do any good in our experiment at large strides (when data locality is low).

This simple experiment also makes clear why database systems are quickly constrained by memory access, even on simple tasks like scanning, that seem to have an access pattern that is easily cacheable (sequential). The default physical representation of a tuple is a consecutive byte sequence (a “record”), which must always be accessed by the bottom operators in a query evaluation tree (typically selections or projections). The record byte-width of typical relational tables is measured in the hundreds of bytes. Figure 6.3 makes quite clear that such large strides lead to worst-case performance, such that the memory access bottleneck kills all CPU performance advances.

To improve performance, we strongly recommend using vertically fragmented data structures. In our Monet system, we fully decompose relational tables on all columns, storing each in a separate Binary Association Tables (BAT). This approach is known in literature as the Decomposed Storage Model [CK85]. A BAT is represented in memory as an array of fixed-size two-field records \([oid, value]\) – called Binary UNits (BUN) – where the oid-s are used to link together the tuples that are decomposed across different BATs. Full vertical fragmentation keeps the database records thin (8 bytes or less) and is therefore the key for reducing memory access cost (staying on the left side of the graphs in Figure 6.3).

### 6.3.3 Calibrator Tool

In order to analyze the impact of memory access cost in detail, we need to know the characteristic parameters of the memory system, like memory sizes, cache sizes, cache line sizes, and access latencies. Often, not all these parameters are (correctly) present

\(^2\)In one memory fetch, the Origin2000 gets 128 bytes, whereas the Sun LX gets only 16; an improvement of factor 8.
in the hardware manual or from the vendor. Thus, we need to calibrate them ourselves. In the following, we describe a simple but powerful **calibration tool** to measure the (cache) memory characteristics of an arbitrary machine.

### Calibrating the Memory System

Our calibrator is a simple C program, mainly a small loop that executes a million memory reads. By changing the stride and the size of the memory area, we force varying cache miss rates. Thus, we can calculate the latency for a cache miss by comparing the execution time without misses to the execution time with exactly one miss per iteration. This approach only works, if memory accesses are executed purely sequential, i.e. we have to make sure, that neither two or more load instructions nor memory access and pure CPU work overlap. We use a simple pointer chasing mechanism to achieve this: the memory area we access is initialized such that each load returns the address for the subsequent load in the next iteration. Thus, modern super-scalar CPUs cannot benefit from their ability to hide memory access latency by speculative execution.

To measure the cache characteristics, we run our experiment several times, varying the stride and the array size. We make sure, that the stride varies at least between 4 byte and twice the maximal expected cache line size, and that the array size varies from half the minimal expected cache size to at least ten times the maximal expected cache size. The leftmost plot in Figure 6.4 depicts the resulting execution time (in nanoseconds) per iteration for different array sizes on our Origin2000 (MIPS R10000, 250 MHz = 4ns per cycle). Each curve represents a different stride. From this figure, we can derive the desired parameters as follows: Up to an array size of 32 KB, one iteration takes 8 nanoseconds (i.e. 2 cycles), independent on the stride. Here, no cache misses occur once the data is loaded, as the array completely fits in L1 cache. One of the two cycles accounts for executing the load instruction, the other one accounts

![Figure 6.4: Calibrator Tool: Cache sizes, lines sizes, and latencies](image-url)
for the latency to access data in L1. With array sizes between 32 KB and 4 MB, the array exceeds L1, but still fits in L2. Thus, L1 misses occur. The miss rate (i.e. the number of misses per iteration) depends on the stride \( s \) and the L1 cache line size \( (L_S_{L1}) \). With \( s < L_S_{L1} \), \( \frac{s}{L_S_{L1}} \) L1 misses occur per iteration (or one L1 miss occurs every \( \frac{L_S_{L1}}{s} \) iterations). With \( s \geq L_S_{L1} \), each load causes an L1 miss.

Figure 6.4 shows that the execution time increases with the stride, up to a stride of 32. Then, it stays constant. Hence, L1 line size is 32 byte. Further, L1 miss latency (i.e. L2 access latency) is 32ns – 8ns = 24ns, or 6 cycles. Similarly, when the array size exceeds L2 size (4 MB), L2 misses occur. Here, L2 line size is 128 byte, and L2 miss latency (memory access latency) is 432ns – 32ns = 400ns, or 100 cycles. Analogously, the middle and the rightmost plot in Figure 6.4 show the results for a Sun Ultra (Sun UltraSPARC [GBC+95] 200 MHz = 5ns per cycle) and an Intel PC (Inte PentiumIII [Die99, KP99] 450 MHz = 2.22ns per cycle).

The **sequential memory bandwidth** for our systems, listed in Table 6.1, is computed from the cache line sizes and the latencies as follows:

\[
bw_{seq} = \frac{L_S_{L2}}{L_S_{L1}} \frac{L_S_{L2}}{l_{Mem}} + \frac{L_S_{L2}}{l_{Mem}} \frac{L_S_{L2}}{L_S_{L1}} + l_{L2}
\]

We will discuss **parallel memory bandwidth** in the next section.

**Calibrating TLB** We use a similar approach as above to measure **TLB miss cost**. The idea here is to force one TLB miss per iteration, but to avoid any cache misses. We force TLB misses by using a stride that is equal to or larger than the systems page size, and by choosing the array size such that we access more distinct spots than there are TLB entries. Cache misses will occur at least as soon as the number of spots accessed exceeds the number of cache lines. We cannot avoid that. But even with less spots accessed, two or more spots might be mapped to the same cache line, causing conflict misses. To avoid this, we use strides that are not exactly powers of two, but slightly bigger, shifted by L2 cache line size, i.e. \( s = 2^x + L_S_{L2} \).

Figure 6.5 shows the results for our three machines. The x-axis now gives the number of spots accessed, i.e. array size divided by stride. Again, each curve represents a different stride. From the leftmost plot (Origin2000), e.g., we derive the following: Like above, we observe the base line of 8 nanoseconds (i.e. 2 cycles) per iteration. The smallest number of spots where the performance decreases due to TLB misses is 64, hence, there are 64 TLB entries. The decrease at 64 spots occurs with strides of 32KB or more, thus, the page size is 32KB. Further, TLB miss latency is 236ns – 8ns = 228ns, or 57 cycles. In the rightmost plot, the second step at 512 spots is caused by L1 misses as L1 latency is 4 times higher than TLB latency on the PC. On the Origin2000 and on the Sun, L1 misses also occur with more than 1024 spots access, but their impact is negligible as TLB latency is almost 10 times higher than L1 latency on these machines.

Table 6.1 gathers the results for all three machines. The PC has the highest L2 latency, probably as its L2 cache is running at only half the CPU’s clock speed, but it has the lowest memory latency and an incredibly low TLB miss latency. The Ori-
gin2000 has the highest memory latency, but due to its large cache lines, it achieves the best sequential memory bandwidth.

### 6.3.4 Parallel Memory Access

It is interesting to note that the calibrated latencies in Table 6.1 do not always confirm the suggested latencies in the sequential scan experiment from Figure 6.3. For the PentiumIII, the access cost per memory read of 52ns at a stride of 32 bytes, and 204ns at a stride of 128 bytes for the Origin2000, are considerably lower than their memory latencies (135ns resp. 424ns), where in the case of the Sun Ultra, the scan measurement at L2 line size almost coincides with the calibrated memory latency.

<table>
<thead>
<tr>
<th>OS</th>
<th>SGI Origin2000</th>
<th>Sun Ultra</th>
<th>Intel PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>MIPS R10000 250 MHz</td>
<td>Sun UltraSparc 200 MHz</td>
<td>Intel PentiumIII 450 MHz</td>
</tr>
<tr>
<td>main memory size</td>
<td>48 GB (4 GB local)</td>
<td>512 MB</td>
<td>512 MB</td>
</tr>
<tr>
<td>L1 cache size</td>
<td>[L1]</td>
<td>32 KB</td>
<td>16 KB</td>
</tr>
<tr>
<td>L1 cache line size</td>
<td>L1 _L1</td>
<td>32 bytes</td>
<td>16 bytes</td>
</tr>
<tr>
<td>L1 cache lines</td>
<td>[L1] _L1</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>[L2]</td>
<td>4 MB</td>
<td>1 MB</td>
</tr>
<tr>
<td>L2 cache line size</td>
<td>L2 _L2</td>
<td>128 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L2 lines</td>
<td>L2 _L2</td>
<td>32.768</td>
<td>64</td>
</tr>
<tr>
<td>TLB entries</td>
<td>TLB</td>
<td>64</td>
<td>8 KB</td>
</tr>
<tr>
<td>page size</td>
<td>[PG]</td>
<td>32 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>TLB size</td>
<td>[TLB]</td>
<td>2 MB</td>
<td>512 KB</td>
</tr>
<tr>
<td>L1 miss latency</td>
<td>L1 _L2</td>
<td>24 ns = 6 cycles</td>
<td>30 ns = 6 cycles</td>
</tr>
<tr>
<td>L2 miss latency</td>
<td>L2 _L2</td>
<td>100 cycles</td>
<td>195 ns = 39 cycles</td>
</tr>
<tr>
<td>TLB miss latency</td>
<td>TLB</td>
<td>57 cycles</td>
<td>270 ns = 54 cycles</td>
</tr>
<tr>
<td>seq. memory bandwidth</td>
<td>bwseq</td>
<td>246 MB/s</td>
<td>193 MB/s</td>
</tr>
<tr>
<td>par. memory bandwidth</td>
<td>bwpar</td>
<td>555 MB/s</td>
<td>244 MB/s</td>
</tr>
</tbody>
</table>

Table 6.1: Calibrated Performance Characteristics
The discrepancies are caused by parallel memory access that can occur on CPUs that feature both speculative execution and a non-blocking memory system. This allows a CPU to execute multiple memory load instructions in parallel, potentially enhancing memory bandwidth above the level of cache-line size divided by latency. Prerequisites for this technique are a bus system with excess transport capacity and a non-blocking cache system that allows multiple outstanding cache misses.

To answer the question what needs to be done by an application programmer to achieve these parallel memory loads, let us consider a simple programming loop that sums an array of integers. Figure 6.6 shows three implementations, where the leftmost column contains the standard approach that results in sequential memory loads into the buf[size] array. An R10000 processor [ZLT196, Yea96] can continue executing memory load instructions speculatively until four of them are stalled. In this loop that will indeed happen if buf[i],buf[i+1],buf[i+2] and buf[i+3] are not in the (L2) cache. However, due to the fact that our loop accesses consecutive locations in the buf array, these four memory references request the same 128-byte L2 cache line. Consequently, no parallel memory access takes place. If we assume that this loop takes 2 cycles per iteration\(^3\), we can calculate that 32 iterations cost 32*2 + 124 = 190 cycles (where 124 is the memory latency on our Origin2000); a total mean cost of 5.94 cycles per addition.

Parallel memory access can be enforced by having one loop that iterates two cursors through the buf[size] array (see the middle column of Figure 6.6). This causes 2 parallel 128 byte (=32 integer) L2 cache line fetches from memory per 32 iterations, for a total of 64 additions. On the R10000, the measured maximum memory bandwidth of the bus is 555MB/s, so fetching two 128-byte cache lines in parallel costs only 112 cycles (instead of 124 + 124). The mean cost per addition is hence 2 + 112/64 = 3.75 cycles.

It is important to note that parallel memory access is achieved only if the ability of the CPU to execute multiple instructions speculatively spans multiple memory references in the application code. In other words, the parallel effect disappears if there is too much CPU work between two memory fetches (more than 124 cycles on the R10000) or if the instructions are too much interdependent, causing a CPU stall before reaching the next memory reference. For database algorithms this means that random access operations like hashing will not profit from parallel memory access, as following a link list (hash bucket chain) causes one iteration to depend on the previous; hence a memory miss will block execution. Only iterative algorithms with independent iterations and CPU processing cost per iteration that is less than the memory latency, will profit, like in the simple scan experiment from Figure 6.3. This experiment reaches optimal parallel bandwidth when the stride is equal to this L2 cache line size. As each loop

\(^3\)As each iteration of our loop consists of a memory load (buf[i]), an integer addition (of “total” with this value), an integer increment (of i), a comparison, and a branch, the R10000 manual suggests a total cost of minimally 6 cycles. However, due to the speculative execution in the R10000 processor, this is reduced to 2 cycles on the average

<table>
<thead>
<tr>
<th>normal loop</th>
<th>multi-cursor</th>
<th>prefetch</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for(int tot=0; i&lt;N; i++)</code> { <code>for(int tot0=tot1==0; i&lt;N/2; i&lt;C; i++)</code> { <code>for(int tot=0; i&lt;N; i++)</code> { <code>tot += buf[i];</code> <code>tot0 += buf[i];</code> <code>tot0 += buf[i+C];</code> <code>tot += buf[i];</code> } } }`</td>
<td><code>int int = tot0 + tot1;</code></td>
<td><code>3.75 cycles/addition</code></td>
</tr>
</tbody>
</table>

Figure 6.6: Three ways to sum an int-array, and their cycles per addition (Origin2000)
CHAPTER 6. MEMORY/CPU OPTIMIZED QUERY PROCESSING

iteration then requests one subsequent cache line, modern CPUs will have multiple memory loads outstanding, executing them in parallel. Results are summarized at the bottom of Table 6.1, showing the parallel effect to be especially strong on the Origin2000 and the PentiumIII. In other words, if the memory access pattern is not sequential (like in equi-join), the memory access penalty paid on these systems is actually much higher than suggested by Figure 6.3, but determined by the latencies from Table 6.1.

6.3.5 Prefetched Memory Access

Computer systems with a non-blocking cache can shadow memory latency by performing a memory fetch well before it is actually needed. CPUs like the MIPS R10000, the Intel PentiumIII, and the newer Sun UltraSPARC II [GT96] processors have special *prefetching instructions* for this purpose. These instructions can be thought of as memory load instructions that do not deliver a result. Their only side-effect is a modification of the status of the caches. The work in [Mow94] studies compiler techniques to generate these prefetching instructions automatically. These techniques optimize array accesses from within loops when most loop information and dependencies are statically available, and as such are very appropriate for scientific codes written in FORTRAN. Database codes written in C/C++, however, do not profit from these techniques as even the most simple table scan implementation will typically result in a loop with both a dynamic stride and length, as these are (dynamically) determined by the width and length of the table that is being scanned. Also, if table values are compared or manipulated within the loop using a function call (e.g., comparing two values for equality using a C function looked up from some ADT table, or a C++ method with late binding), the unprotected pointer model of the C/C++ languages forces the compiler to consider the possibility of side effects from within that function; eliminating the possibility of optimization.

As a way around such situations, the MipsPRO compiler for the R10000 systems of Silicon Graphics allows the programmer to pass explicit prefetching hints by use of pragma’s, as depicted in the rightmost column of Figure 6.6. This pragma tells the compiler to request the next cache line once in every 32 iterations. Such a prefetch-frequency is generated by the compiler by applying loop unrolling (it unrolls the loop 32 times and inserts one prefetch instruction). By hiding the memory prefetch behind 64 cycles of work, the mean cost per addition in this routine is reduced to \( 2 + \frac{(124-64)}{32} = 3.88 \) cycles. Optimal performance is achieved in this case when prefetching two cache lines ahead every 32 iterations (*#prefetch buf[i+64] freq=32*). The 124 cycles of latency are then totally hidden behind 128 cycles of CPU work, and a new cache line is asked every 64 cycles. This setting effectively combines prefetching with parallel memory access (two cache lines in 128 cycles instead of 248), and reduces the mean cost per addition to the minimum 2 cycles; three times faster than the simple approach!

6.3.6 Future Hardware Features

In spite of memory latency staying constant, hardware manufacturers have been able to increase memory bandwidth in line with the performance improvements of CPUs, by working with ever wider lines in the L1 and L2 caches. As cache-lines grew wider,
buses also did. The latest Sun UltraII workstations, for instance, have a 64-byte L2 cache-line which is filled in parallel using a PCI bus of 576 bits wide (576 = 64*8 plus 64 bits overhead). The strategy of doubling memory bandwidth by doubling the number of DRAM chips and bus lines is now seriously complicating system board design. The future Rambus [Ram96] memory standard eliminates this problem by providing an “protocol-driven memory bus”. Instead of designating one bit in the bus for one bit of data transported to the cache-line, this new technology serializes the DRAM data into packets using a protocol and sends these packets over a thin (16-bit) bus that runs at very high speeds (up to 800MHz). While this allows for continued growth in memory bandwidth, it does not provide the same perspective for memory latency, as it is still DRAM that is being accessed, and there will still be the relatively long distance for the signals to travel between the CPU and the memory modules on the system board; both factors ensuring a fixed startup cost (latency) for any memory traffic.

An interesting proposal worth mentioning here has been “smarter memory” [MKW+98], which would allow the programmer to give a “cache-hint” by specifying the access pattern that is going to be used on a memory region in advance. In this way, the programmer is no longer obliged to organize his data structures around the size of a cache line, rather lets the cache adapt its behavior to the needs of the application. Such a configurable system is in some sense a protocol-driven bus system, so Rambus is a step in this direction. Configurable memory access has not yet been considered for custom hardware, however, let alone in OS and compiler tools that would need to provide the possibility to incorporate such hints for user-programs.

In a development that started from 2000, however, the memory subsystems for AMD PC platforms (the DDR SDRAM-based NVidia Nforce chipset [NV101]), as well as the Pentium 4 [HSU+01], AMD Athlon XP [Adv01] and Sun SPARC III [LH99] processors themselves, do implement “hardware prefetching”. In all cases, this entails a simple algorithm that detects pure consecutive memory access, and in those cases automatically issues memory prefetch requests. In a sense, this is a way to exploit the available memory bandwidth, which is an alternative to the traditional measure of just increasing the cache line width (which has the disadvantage that the cache becomes coarse-grained, hence less efficient for random-access loads). If the algorithms prefetches sufficient cache lines ahead, this new hardware feature has the potential of eliminating all memory access cost from algorithms that exhibit pure sequential access (at a pace lower than the maximum memory bandwidth) automatically.

Concerning CPU technology, it is anticipated [Sem97] that the performance advances dictated by Moore’s law will continue well into at least 2010. However, performance increase will also be brought by more parallelism within the CPU. The upcoming IA-64 architecture has a design called Explicitly Parallel Instruction Computing (EPIC) [ACM+98, SA00], which allows instructions to be combined in bundles, explicitly telling the CPU that they are independent. The IA-64 is specifically designed to be scalable in the number of functional units, so while newer versions are released, more and more parallel units will be added. This means that while current PC hardware depends relatively less on parallel CPU execution than the RISC systems, this will most probably change in the new 64-bit PC generation.

Another ongoing CPU trend is tied to the trend of ever higher CPU clock speeds. The simplest way to increase CPU speed is to use more advanced (i.e., smaller) silicon process technology. The trend of process technology decreasing the transistor gate length with 50% every 2 years is one of the basic enablers of Moore’s law. On top of
that, though, hardware designers tend to increase the length of the processor pipeline in every new design. For example, the MIPS R10K has 5 stages, the Pentium III has 7, the AMD Athlon has 11, and the Intel Pentium 4 has 20 [ZLT96, Yea96, Die99, KP99, Adv99, Adv00, HSU+01]! The reason for doing so is that by splitting up the work of instruction execution in the CPU into more stages, each individual stage has to do less work, completes faster, making possible a higher clock speed on the same process. The side-effect of this design strategy is that new processors depend ever more strongly on speculative execution. In other words, if such processors have to execute difficult-to-predict code (i.e., non-Monet database code), the performance price paid for branch mis-predictions (i.e., flushing the pipeline) becomes ever greater.

A final trend in CPU design to be mentioned here is multiprocessing-on-a-chip. The simplest form, found in the latest IBM Power4 [TDF01] and Sun MAJC [Sun99] chips, is to simply place two identical CPUs on the same chip, together with SMP cache coherency logic between them. The motivation for doing so is to allow more efficient cache coherency, and – more importantly – simply because the ever-shrinking process technology sizes give CPU designers such a huge “transistor budget” (i.e., currently in the order of hundreds of millions) that they are losing the race of thinking up designs in time for effectively using such huge amounts of transistors. Putting multiple incarnations of a smaller existing CPU design on one chip is a way to sensibly efficiently use current chip manufacturing possibilities. A more subtle approach to multiprocessing-on-a-chip is called Simultaneous Multi-Threading (SMT) [EEL+97]. Here, one observes that the currently executing thread of “difficult” code often starves due to branch mis-predictions and memory wait cycles, while the OS might have other threads ready, waiting to execute (e.g. as in an SMP parallel database, executing a query on a single-CPU machine). By adding only a small bit of extra logic (i.e. replicating the register sets and some memory management support), one could make a CPU capable of executing 2, 4 or more threads simultaneously. Therefore, SMT is a more intelligent and efficient way of increasing performance than simply putting identical CPUs on the same chip (which replicates full, poorly used, CPUs). SMT will appear in 2002 in the new Intel Pentium 4 Xeon [Int01], with other hardware manufacturers to follow. As a multiprocessing-on-a-chip CPU (be it SMT or not) essentially makes for an SMP machine, this ongoing trend means that in the long term parallel processing will become ubiquitous (finally), in the sense that any machine will be an SMP machine (at least).

Summarizing, we have identified the following ongoing trends in modern hardware:

- CPU performance keeps growing with Moore’s law for years to come.
- A growing part of this performance increase will come from parallelism within the chip.
- New bus technology will provide sufficient growth in memory bandwidth.
- Memory latency will not improve significantly.

This means that the failure of current DBMS technology in properly utilizing memory and CPU resources of modern hardware [ADHW99, KPH*98, BGB98, TLPZT97] will not go away by itself, rather will grow worse. Modern database architecture should therefore take into account this new hardware environment. With this motivation, we investigate in the following new approaches to large main memory equi-joins. That specifically aim at optimizing resource utilization of modern hardware.
6.4. PARTITIONED HASH-JOIN

Shatdahl et al. [SKN94] showed that a main-memory variant of Grace Join, in which both relations are first partitioned on hash-number into $H$ separate clusters, such that each fit the memory cache, performs better than normal bucket-chained hash join. This work employs a straightforward clustering-algorithm that simply scans the relation to be clustered once, inserting each scanned tuple in one of the clusters, as depicted in Figure 6.7. This constitutes a random access pattern that writes into $H$ separate locations. If this $H$ is too large, there are two factors that degrade performance. First, if $H^4$ exceeds the number of TLB entries each memory reference will become a TLB miss. Second, if $H$ exceeds the number available cache lines (L1 or L2), cache trashing occurs, causing the number of cache misses to explode.

As an improvement over this straightforward algorithm, we propose a clustering algorithm that has a memory access pattern that requires less random-access, even for high values of $H$.

6.4.1 Radix-Cluster Algorithm

The Radix-Cluster algorithm divides a relation into $H$ clusters using multiple passes (see Figure 6.8). Radix-Clustering on the lower $B$ bits (called the “Radix-Bits”) of the integer hash-value of a column is achieved in $P$ sequential passes, in which each pass clusters tuples on $B_p$ bits, starting with the leftmost Radix-Bits ($\sum_{i=1}^{P} B_p = B$).

The number of clusters created by the Radix-Cluster is $H = \prod_{i}^{P} H_p$, where each pass subdivides each cluster into $H_p = 2^{B_p}$ new ones. When the algorithm starts, the entire relation is considered as one single cluster, and is subdivided in $H_1 = 2^{B_1}$ clusters.

4If the relation is very small and fits the total number of TLB entries times the page size, multiple clusters will fit into the same page and this effect will not occur.
The next pass takes these clusters and subdivides each in $H_2 = 2^{B_2}$ new ones, yielding $H_1 * H_2$ clusters in total, etc. Note that with $P = 1$, Radix-Cluster behaves like the straightforward algorithm.

For ease of presentation, we did not use a hash function in the table of integer values displayed in Figure 6.8. In practice, though, it is better to use such a function even on integers in order to ensure that all bits of the table values play a role in the lower bits of the radix number.

The interesting property of the Radix-Cluster is that the number of randomly accessed regions $H_x$ can be kept low; while still a high overall number of $H$ clusters can be achieved using multiple passes. More specifically, if we keep $H_x = 2^{B_2}$ smaller than the number of cache lines and the number of TLB entries, we totally eliminate both TLB and cache trashing.

After Radix-Clustering a column on $B$ bits, all tuples that have the same $B$ lowest bits in its column hash-value, appear consecutively in the relation, typically forming chunks of $C/2^B$ tuples (with $C$ denoting the cardinality of the entire relation). It is therefore not strictly necessary to store the cluster boundaries in some additional data structure; an algorithm scanning a Radix-Clustered relation can determine the cluster boundaries by looking at these lower $B$ Radix-Bits. This allows very fine clusterings without introducing overhead by large boundary structures. It is interesting to note that a Radix-Clustered relation is in fact ordered on Radix-Bits. When using this algorithm in the Partitioned Hash-Join, we exploit this property, by performing a merge step on the Radix-Bits of both Radix-Clustered relations to get the pairs of clusters that should be Hash-Joined with each other.

### 6.4.2 Quantitative Assessment

The Radix-Cluster algorithm presented in the previous section provides three tuning parameters:

1. the number of Radix-Bits used for clustering ($B$), implying the number of clusters $H = 2^B$,
2. the number of passes used during clustering ($P$),
3. the number of Radix-Bits used per clustering pass ($B_p$).

In the following, we present an exhaustive series of experiments to analyze the performance impact of different settings of these parameters. After establishing which parameters settings are optimal for Radix-Clustering a relation on $B$ Radix-Bits, we turn our attention to the performance of the join algorithm with varying values of $B$. For both phases, clustering and joining, we investigate, how appropriate implementation techniques can improve the performance even further. Finally, these two experiments are combined to gain insight in overall join performance.

### Experimental Setup

In our experiments, we use binary relations (BATs) of 8 bytes wide tuples and varying cardinalities ($C$), consisting of uniformly distributed random numbers. Each value occurs three times. Hence, in the join-experiments, the join hit-rate is three. The result of a join is a BAT that contains the [oid, oid] combinations of matching tuples (i.e.,
6.4. PARTITIONED HASH-JOIN

<table>
<thead>
<tr>
<th>category</th>
<th>MIPS R10K</th>
<th>Sun UltraSPARC</th>
<th>Intel Pentiumml</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory access</td>
<td>L1.data.misses * 6 cy</td>
<td>STALL_LOAD</td>
<td>cycles_while_DCU_miss.outstanding</td>
</tr>
<tr>
<td></td>
<td>L2.data.misses * 100 cy</td>
<td>STALL_STORBUF MTLB * 54 cy</td>
<td>MTLB * 5 cy</td>
</tr>
<tr>
<td></td>
<td>TLB.misses * 57 cy</td>
<td>STALL_IC_MISS</td>
<td>cycles_instruction_fetch_pipe.is_stalled</td>
</tr>
<tr>
<td></td>
<td>L1.inst.misses * 6 cy</td>
<td></td>
<td>ITLB.misses * 32 cy</td>
</tr>
<tr>
<td></td>
<td>L2.inst.misses * 100 cy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU stalls</td>
<td>branches.mis-predicted * 4 cy</td>
<td>STALL_MISPRED STALL_FPDEP</td>
<td>taken.mis-predicted_branches.retired * 17 cy</td>
</tr>
<tr>
<td>integer divisions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real CPU</td>
<td>“the rest”</td>
<td>“the rest”</td>
<td>“the rest”</td>
</tr>
</tbody>
</table>

Table 6.2: Hardware Counters used for Execution Time Breakdown

a join-index [Val87]). Just like in [SKN94], we do not include tuple reconstruction in our comparison. The issues arising from taking into account the cost of projecting columns from both join input relations into the output are later described in detail in Section 6.5. The experiments were carried out on the machines presented in Section 6.3.3, an SGI Origin2000, a Sun Ultra, and an Intel PC (cf. Table 6.1).

To analyze the performance behavior of our algorithms in detail, we breakdown the overall execution time into the following major cost categories:

memory access In addition to memory access costs for data as analyzed above, these also contain memory access costs caused by instruction cache misses.

CPU stalls Beyond memory access, there are other events that make the CPU stall, like branch mis-predictions or other so-called resource related stalls.

divisions We treat integer divisions separately, as they play a significant role in our Hash-Join (see below).

real CPU The remaining time, i.e. the time, the CPU is indeed busy, executing the algorithms.

The three architectures we investigate, provide different hardware counters [BZ98] that enable us to measure each of these cost factors accurately. Table 6.2 gives an overview of the counters used. Some counters yield the actual CPU cycles spent during a certain event, others just return the number of events that occurred. In the latter case, we multiply the counters by the penalties of the events (as calibrated in Section 6.3.3). None of the architectures provides a counter for the pure CPU activity. Hence, we subtract the cycles spent on memory access, CPU stalls, and integer division from the overall number of cycles and assume the rest to be pure CPU cost.

In our experiments, we discovered that in our algorithms, branch mis-predictions and instruction cache misses do not play a role on either architecture. Thus, for simplicity of presentation, we omit them in our further considerations.

5 Taken from [ADHW99].

6 This counter originally includes “cycles_while_DCU_miss.outstanding”. We use only the remaining part after subtracting “cycles_while_DCU_miss.outstanding”, here.
Radix-Cluster

To analyze the impact of all three parameters \((B, P, B_p)\) on Radix-Clustering, we conduct two series of experiments, keeping one parameter fixed and varying the remaining two.

First, we conduct experiments with various numbers of Radix-Bits and passes, distributing the Radix-Bits evenly across the passes. Figure 6.9 shows an execution time breakdown for 1-pass Radix-Cluster \((C = 8M)\) on each architecture. The pure CPU costs are nearly constant across the all numbers of Radix-Bits, taking about 3 seconds on the Origin, 2.5 seconds on the PC, and a about 5.5 seconds on the Sun. Memory and TLB costs are low with small numbers of Radix-Bits, but grow significantly with rising numbers of Radix-Bits. With more than 6 Radix-Bits, the number of clusters to be filled concurrently exceeds the number of TLB entries \((64)\), causing the number of TLB misses to increases tremendously. On the Origin and on the Sun, the execution time increases significantly due to their rather high TLB miss penalties. On the PC however, the impact of TLB miss misses is hardly visible due to its very low TLB miss penalty. Analogously, the memory costs increase as soon as the number of clusters exceeds the number of L1 and L2 cache lines, respectively. Further, on the PC, “resource related stalls” (i.e. stalls due to functional unit unavailability) play a significant role. They make up one fourth of the execution time when the memory costs are low. When the memory costs rise, the resource related stalls decrease and finally vanish completely, reducing the impact of the memory penalty. Or, in other words, minimizing the memory access costs does not fully pay back, as the resource related stalls partly take over their part.

Figure 6.10 depicts the breakdown for Radix-Cluster using the optimal number of passes. The idea of multi-pass Radix-Cluster is to keep the number of clusters generated per pass low—and thus the memory cost—at the expense of increased CPU cost. Obviously, the CPU cost are too high to avoid the TLB cost by using two passes with more than 6 Radix-Bits. Only with more than 15 Radix-Bits, i.e. when the memory cost exceed the CPU cost, two passes win over one pass.

The only way to improve this situation is to reduce the CPU cost. Figure 6.11 shows the source code of our Radix-Cluster routine. It performs a single-pass clustering on the \(D\) bits that start \(R\) bits from the right (multi-pass clustering in \(P > 1\) passes on \(B = P \times D\) bits is done by making subsequent calls to this function for \(p = 1\) through \(p = P\) with parameters \(D_p = D\) and \(R_p = (p - 1) \times D\), starting with the input relation and using the output of the previous pass as input for the next). As the algorithm itself is already very simple, improvement can only be achieved by means of implementation techniques. We replaced the generic ADT-like implementation by a specialized one for each data type. Thus, we could inline the hash function and replace the radix by a simple assignment, saving two function calls per iteration.

Figure 6.12 shows the execution time breakdown for the optimized 1-pass Radix-Cluster.

Figure 6.13 shows the execution time breakdown for the optimized multi-pass Radix-Cluster. The CPU cost has reduced tremendously, by almost factor 4. Replacing the two function calls has two effects. First, some CPU cycles are saved. Second, the CPUs can benefit more from the internal parallel capabilities using speculative execution, as the code has become simple and more predictable. On the PC, the resource stalls have doubled, neutralizing the CPU improvement partly. Probably, the simple loop does not offer enough “meat” to fill the pipelines efficiently.
6.4. PARTITIONED HASH-JOIN

Figure 6.9: Radix-Cluster ($C = 8M$, 1 pass)

Figure 6.10: Radix-Cluster ($C = 8M$, best)
CHAPTER 6. MEMORY/CPU OPTIMIZED QUERY PROCESSING

```c
#define HASH(v) ((v>>7) XOR (v>>13) XOR (v>>21))
typedef struct {
  int v1,v2; /* simplified binary tuple */
} bun;

define RadixCluster(bun *rel, bun *rel_end, /* input relation */
  int R, int D /* radix and cluster bits */) {
  int M = (2^D - 1) << R;
  for(bun *cur = rel; cur < rel_end; cur++) {
    int idx = HASH(cur->v2) & M;
    memcpy(dst[idx], cur, sizeof(bun));
    if (++dst[idx] > dst_end[idx]) REALLOC(dst[idx], dst_end[idx]);
  }
}
```

Figure 6.11: C language Radix-Cluster with annotated CPU optimizations (right)

With this optimization, multi-pass Radix-Cluster is feasible already with smaller numbers of Radix-Bits. On the Origin, two passes win with more than 6 Radix-Bits, and three passes win with more than 13 Radix-Bits, thus avoiding TLB trashing nearly completely. On the PC, the improvement is marginal. The severe impact of resource stalls with low numbers of Radix-Bits makes the memory optimization of multi-pass Radix-Cluster almost ineffective.

In order to estimate the performance of Radix-Cluster, and especially to predict the number of passes to be used for a certain number of Radix-Bits, we now provide an accurate cost model for Radix-Cluster. The cost model takes the number of passes, the number of Radix-Bits, and the cardinality as input and estimates the number of memory related events, i.e. L1 cache misses, L2 cache misses, and TLB misses. The overall execution time is calculated by scoring the events with their penalties and adding the pure CPU costs.

\[
T_c(P,B,C) = P \left( C \cdot w_c + M_{L1,c} \left( \frac{B}{P} \right)^* \cdot l_{L2} + M_{L2,c} \left( \frac{B}{P} \right)^* \cdot l_{mem} + M_{TLB,c} \left( \frac{B}{P} \right)^* \cdot l_{TLB} \right) \\
\]

with

\[
M_{L1,c}(B_p,C) = 2 \cdot \frac{\text{Re}|L_i|}{L_i} + \left\{ \begin{array}{l}
C \cdot \frac{\text{H}_{r}}{\frac{\text{Re}|L_i|}{L_i}} \cdot \min\left\{ 1, \frac{\text{Re}|L_i|}{L_i} \right\}, \text{ if } \{\text{H}_{r}, \text{Re}|L_i|\} \leq \{L_i|L_i| \}
C \cdot \min\left\{ 3,1 + \log\left( \frac{\text{Re}|L_i|}{L_i} \right) \right\}, \text{ if } \{\text{H}_{r}, \text{Re}|L_i|\} > \{L_i|L_i| \}
\end{array} \right.
\]

\[
M_{L2,c}(B_p,C) = 2 \cdot \frac{\text{Re}|P_g|}{P_g} + \left\{ \begin{array}{l}
C \cdot \left( 1 - \frac{\text{TLB}}{\frac{\text{Re}|P_g|}{P_g}} \right), \text{ if } \{\text{H}_{r}, \text{Re}|P_g|\} \leq \{\text{TLB} \}
C \cdot \left( \frac{\text{H}_{r}}{\frac{\text{Re}|P_g|}{P_g}} \right), \text{ if } \{\text{H}_{r}, \text{Re}|P_g|\} > \{\text{TLB} \}
\end{array} \right.
\]

\[
\text{(if } \{\text{Re}|P_g| > \{\text{TLB}\} \text{)} + \left\{ \begin{array}{l}
C \cdot \min\left\{ 2,1 + \log\left( \frac{\text{H}_{r}}{\frac{\text{Re}|P_g|}{P_g}} \right) \right\}, \text{ if } \text{H}_{r} \leq \{L_2|L_2\}
C \cdot \min\left\{ 2,1 + \log\left( \frac{\text{H}_{r}}{\frac{\text{Re}|P_g|}{P_g}} \right) \right\}, \text{ if } \text{H}_{r} > \{L_2|L_2\}
\end{array} \right.
\]

\[
\text{Re}|L_i| \text{ and } |Cl|_L_i \text{ denote the number of cache lines per relation and cluster, respectively, } \text{Re}|P_g| \text{ the number of pages per relation, } |L_i|_L_i \text{ the total number of cache lines, both for the L1 } (i = 1) \text{ and L2 } (i = 2) \text{ caches, and } |\text{TLB}| \text{ the number of TLB entries. } w_c \text{ denotes the pure CPU cost per tuple. To calibrate } w_c, \text{ we reduced the cardinality so that all data fits in L1, and pre-loaded the input relation. Thus, we avoided memory}
\]
6.4. PARTITIONED HASH-JOIN

access completely. We measured \( w_c = 100\text{ns} \) on the Origin2000, \( w_c = 200\text{ns} \) on the Sun, and \( w_c = 180\text{ns} \) on the PC (including resource stalls).

The first term of \( M_{L_i,c} \) equals the minimal number of \( L_i \) misses per pass for fetching the input and storing the output. The second term counts the number of additional \( L_i \) misses, when the number of distinct \( L_i \) lines accessed concurrently (i.e. \( x = \min \{ H_p, |R|_{L_i} \} \))\(^7\) either approaches the number of available \( L_i \) lines (\( x \leq |L_i|_{L_i} \)) or even exceeds this (\( x > |L_i|_{L_i} \)). A detailed description of this and the following formulas is given in [MBK99]. First, the probability that the requested cluster is not in the cache—due to address conflicts—increases until \( H_p = |L_i|_{L_i} \). Then, the cache capacity is exhausted, and a cache miss for each tuple to be assigned to a cluster is certain. But, with further increasing \( H_p \), the number of cache misses also increases, as now also the cache lines of the input may be replaced before all tuples are processed. Thus, each input cache line has to be loaded more than once. The first two terms of \( M_{TLB,c} \) are made up analogously. Additionally, using a similar schema as \( M_{L_i,c} \), the third term models—for relations that contain more pages than there are TLB entries—

\(^7\)Using \( \min \{ H_p, |R|_{L_i} \} \) instead of simply \( H_p \) takes into account, that smaller relations may completely fit in \( L_i \), i.e. with \( H_p > |L_i|_{L_i} > |R|_{L_i} \), several (tiny) clusters share one cache line.
the additional TLB misses that occur when the number of clusters either approaches the number of available L2 lines \((H_p \leq |L2|_{L2})\) or even exceeds this \((H_p > |L2|_{L2})\).

Figure 6.14 compares our model (lines) with the experimental results (points) on the Origin2000 for different cardinalities. The model proves to be very accurate for the number of cache misses (both, L1 and L2) and TLB misses. The predicted elapsed time is also reasonably accurate on all architectures (cf. Figure 6.15). The plots clearly reflect the increase in cache and TLB misses and their impact on the execution time whenever the number of clusters per pass exceeds the respective limits.

Only for very high cardinalities on the Origin2000, it is slightly too low. Here, the complete amount of data to be handled reaches the capacity of a single CPU board. Thus, parts of the allocated memory are likely to be physically located on another board, requiring either remote memory accesses or process migration to the other board. The occurrence of such events is hard to predict, hence, our model does not include the additional costs for them.

The question remaining is how to distribute the number of Radix-Bits over the passes. We conducted another number of experiments, using a fixed number of passes, but varying the number of Radix-Bits per pass. Figure 6.16 depicts the respective results for 4, 8, 12, 16, 20, and 24 Radix-Bits, using 2 passes. The x-axis shows \(B + \frac{B_1}{5}\), hence, for each number of Radix-Bits \((B = B_1 + B_2)\) there is a short line segment consisting of \(B - 1\) points. The first (leftmost) point of each segment represents \(B_1 = 1, B_2 = B - 1\), the last (rightmost) point represents \(B_1 = B - 1, B_2 = 1\). The results show, that even distribution of Radix-Bits \((B_1 \approx B_2 \approx \frac{B}{2})\) achieves the best performance.
6.4. PARTITIONED HASH-JOIN

Isolated Join Performance

We now analyze the impact of the number of Radix-Bits on the pure join performance, not including the clustering cost. With 0 Radix-Bits, the join algorithm behaves like a simple non-partitioned Hash-Join.

The Partitioned Hash-Join exhibits increased performance with increasing number of Radix-Bits. Figure 6.17 shows that this behavior is mainly caused by the memory costs. While the CPU cost is almost independent of the number of Radix-Bits, the memory cost decrease with increasing number of Radix-Bits. The performance increase flattens after the point where the entire inner cluster (including its hash table) consists of less pages than there are TLB entries (64). Then, it also fits the L2 cache comfortably. Thereafter, performance increases only slightly until the point that the inner cluster fits the L1 cache. Here, performance reaches its maximum. The fixed overhead by allocation of the hash-table structure causes performance to decrease when the cluster sizes get too small and clusters get very numerous. Again, the PC shows a slightly different behavior. TLB cost does not play any role, but “partial stalls” (i.e. stalls due to dependencies among instructions) are significant with small numbers of Radix-Bits. With increasing numbers of clusters, the partial stalls decrease, but then, resource stalls increase, almost neutralizing the memory optimization.

Like with Radix-Cluster, once the memory access is optimized, the execution of Partitioned Hash-Join is dominated by CPU cost. Hence, we applied the same optimizations as above. We inlined the hash-function calls during hash build and hash probe as well as the compare-function call during hash probe and replaced two *radix* by simple assignments, saving five function calls per iteration. Further, we replaced the modulo division ("%") for calculating the hash index by a bit operation ("&"). Figure 6.18 depicts the original implementation of our Hash-Join routine and the optimizations.

Figure 6.15: Measured (points) and Modeled (lines) Performance of Radix-Cluster
applied.

Figure 6.19 shows the execution time breakdown for the Optimized Partitioned Hash-Join. For the same reasons as with Radix-Cluster, the CPU cost are reduced by almost factor 4 on the Origin and the Sun, and by factor 3 on the PC. The expensive divisions have vanished completely. Additionally, the dependency stalls on the PC have disappeared, but the functional unit stalls remain almost unchanged.

As for the Radix-Cluster, we also provide a cost model for the Partitioned Hash-Join. The model takes the number of Radix-Bits, the cardinality, and the (average) repeat rate of the join column values (i.e. the join hit rate) as input.

\[ T_h(B, C, r) = C \cdot w_h + M_{L1,h}(B, C, r) \cdot l_{L1} + M_{L2,h}(B, C, r) \cdot l_{Mem} + M_{TLB,h}(B, C, r) \cdot l_{TLB} \]

with

\[ M_{L1,h}(B, C, r) = (2 + r) \cdot |Re|_{Li} + \begin{cases} C \cdot \frac{|CI|}{|Li|}, & \text{if } |CI| \leq |Li| \\ C \cdot (4 + 2r) \cdot \left(1 - \frac{|Li|}{|CI|}\right), & \text{if } |CI| < |Li| \end{cases} \]

and

\[ M_{TLB,h}(B, C, r) = (2 + r) \cdot |Re|_{P_g} + \begin{cases} C \cdot \frac{|CI|}{|TLB|}, & \text{if } |CI| \leq |TLB| \\ C \cdot (4 + 2r) \cdot \left(1 - \frac{|TLB|}{|CI|}\right), & \text{if } |CI| > |TLB| \end{cases} \]

\[ |Re|_{Li}, |Re|_{P_g}, \text{ and } |TLB| \] are as above. \(|CI|, |Li|, \text{ and } |TLB|\) denote (in byte) the cluster size, the sizes of both caches \((i \in \{1, 2\})\), and the memory range covered by \(|TLB|\) pages, respectively.

\(w_h\) represents the pure CPU costs per tuple for building the hash-table, doing the hash lookup and creating the result. We calibrated \(w_h = 600\text{ns}\) on the Origin2000, \(w_h = 1100\text{ns}\) on the Sun, and \(w_h = 711\text{ns}\) on the PC (including resource stalls).

The first term of \(M_{L1,h}\) equals the minimal number of \(Li\) misses for fetching both operands and storing the result. The second term counts the number of additional \(Li\)

\(^8\)For simplicity of presentation, we assume the cardinalities of both input relations to be equal.
misses, when the cluster size either approaches \( L_i \) size or even exceeds this. As soon as the clusters get significantly larger than \( L_i \), each memory access yields a cache miss due to cache trashing: 4 memory accesses per tuple for accessing the outer relation and the bucket array during hash build and hash probe, and 2 memory access per join hit to access the inner relation and the chain-lists. The number of TLB misses is modeled analogously.

Figures 6.20 and 6.21 confirm the accuracy of our model (lines) for the number of L1, L2, and TLB misses on the Origin2000, and for the elapsed time on all architectures.

**Overall Join Performance**

After having analyzed the impact of the tuning parameters on the clustering phase and the joining phase separately, we now turn our attention to the combined cluster and join cost. Radix-cluster gets cheaper for less Radix-Bits, whereas Partitioned Hash-Join gets more expensive. Putting together the experimental data we obtained on both cluster- and join-performance, we determine the optimum number of \( B \) for relation cardinality.

It turns out that there are three possible strategies, which correspond to the diagonals in Figure 6.21:

**phash L2** Partitioned Hash-Join on \( B = \log_2(C \times 12/||L2||) \) clustered bits, so the inner relation plus hash-table fits the L2 cache. This strategy was used in the work of Shatdahal et al. [SKN94] in their Partitioned Hash-Join experiments.

**phash TLB** Partitioned Hash-Join on \( B = \log_2(C \times 12/||TLB||) \) clustered bits, so
# CHAPTER 6. MEMORY/CPU OPTIMIZED QUERY PROCESSING

```c
hash_join(bun *dst, bun *end) /* start and end of result buffer */
  bun *outer, bun *outer_end, bun *inner, bun* inner_end, /* inner and outer relations */
  int R /* radix bits */
} {
  /* build hash table on inner */
  int pos=0, S=inner_end-inner, H=log2(S), N=2^H, M=(N-1)<<R;
  int next[S], bucket[N] = { -1 }; /* hash bucket array and chain-lists */
  for(bun *cur=inner; cur<inner_end; cur++) {
    int idx = ((*hashFcn)(cur->v2) >> R) % N;
    next[pos] = bucket[idx];
    bucket[idx] = pos++;
  }
  /* probe hash table with outer */
  for(bun *cur=outer; cur<outer_end; cur++) {
    int idx = ((*hashFcn)(cur->v2) >> R) % N;
    for(int hit=bucket[idx]; hit>0; hit=next[hit]) {
      if (((compareFcn)(cur->v2, inner[hit].v2)==0)) {
        memcpy(&dst->vl, &cur->vl, sizeof(int));
        memcpy(&dst->v2, &inner[hit].vl, sizeof(int));
        if (dst>end) REALLOC(dst, end);
      }
    }
  }
}

Figure 6.18: C language hash-join with annotated CPU optimizations (right)

Figure 6.19: Optimized Partitioned Hash-Join (C = 8M)
6.4. PARTITIONED HASH-JOIN

Figure 6.20: Measured (points) and Modeled (lines) Events of Partitioned Hash-Join (Origin2000)

Origin2000

Sun Ultra

Intel PC

Figure 6.21: Measured (points) and Modeled (lines) Performance of Partitioned Hash-Join
the inner relation plus hash-table spans at most $|TLB|$ pages. Our experiments show a significant improvement of the pure join performance between phash L2 and phash TLB.

**phash L1** Partitioned Hash-Join on $B = \log_2(C*12/|L1|)$ clustered bits, so the inner relation plus hash-table fits the L1 cache. This algorithm uses more clustered bits than the previous ones, hence it really needs the multi-pass Radix-Cluster algorithm (a straightforward 1-pass cluster would cause cache trashing on this many clusters).

Figure 6.22 shows the overall performance for the original (thin lines) and the CPU-optimized (thick lines) versions of our algorithms, using 1-pass and multi-pass clustering. In most cases, phash TLB is the best strategy, performing significantly better than phash L2. On the Origin2000 and the Sun, the differences between phash TLB and phash L1 are negligible. On the PC, phash L1 performs slightly better than phash TLB. With very small cardinalities, i.e. when the relations do not span more memory pages than there are TLB entries, clustering is not necessary, and the non-partitioned Hash-Join (“simple hash”) performs best.

Further, these results show, that CPU and memory optimization support each other and magnify their effects. The gain of CPU optimization for phash TLB is bigger than that for simple hash, and the gain of memory optimization for the CPU-optimized implementation is bigger than that for the non-optimized implementation. E.g., for large relations on the Origin 2000, CPU optimization reduces the execution time of simple hash by approximately 20%, whereas it yields 66% with phash TLB. Analogously, memory optimization achieves a reduction of slightly less than 60% for the original implementation, but more than 80% for the optimized implementation. Combining both optimizations reduces it by almost 90% (a factor 10 of improvement).
The same can be observed for the Sun Ultra, although the absolute gains are somewhat smaller due to the fact that the Ultra CPU is so slow that trading memory for CPU is not as beneficial as on the Origin2000.

The overall effect of our optimizations on the PentiumIII is just over 50%. One cause of this is the low memory latency on the PC, that limits the gains when memory access is optimized. The second cause is the appearance of the “resource-stalls”; which surge in situations where all other stalls are eliminated (and the RISC chips are really steaming). We expect, though, that future PC hardware with highly parallel IA-64 processors and new Rambus memory systems (that offer high bandwidth but high latencies) will show a more RISC-like performance on our algorithms.

### 6.5 Join Processing With Projections

While the Section 6.4 provides insight in the performance behavior of the various join algorithms, any real-life RDBMS join query goes accompanied by some projection of non-join columns into the result. The precise way in which this is done strongly differs between “traditional” relational query processing and the vertically fragmented query processing in Monet.

In this Section, we investigate optimization of CPU- and memory-resources of equi-join including projections, as formulated by the following SQL join-project query:

```sql
SELECT larger.ai, ..., larger.ay, smaller.b1, ..., smaller.bz
FROM larger, smaller
WHERE larger.key = smaller.key
```

Without loss of genericity, we assume that the “larger” table has the same number of tuples or more than the “smaller” table.

We now discuss what extra query costs are incurred if projection columns are taken into account and discuss algorithms and query processing strategies for optimizing CPU- and memory-resources. In Section 6.5.1 we do this in a “traditional” relational DBMS setting, while in Section 6.5.2 we do the same for Monet, among other by contributing a new cache-conscious query processing algorithm called Radix-Decluster that works in conjunction with the earlier described Radix-Cluster. Finally, in Section 6.5.3 we evaluate the performance of the various query processing strategies.

#### 6.5.1 Cache-Conscious Join: “traditional” Strategy

In “traditional” relational query processing, the simplification of omitting costs of projection is conceptually trivial as the projection column values are just a bit of “extra luggage” traveling with the tuples that flow through an operator tree. This does not change anything to the join algorithm. In the case of Hash-Join, it means that the projection columns of the inner relation are included in the hash-table that is built when scanning the full inner relation. Then, the full outer relation is scanned, hash-lookup performed, and a result table is produced that consists of the projection column values of both tables.

Similarly, when using Radix-Cluster followed by Partitioned Hash-Join in a relational query engine, the first scan of the Radix-Cluster accesses the full relation; producing a clustered copy that includes the key column(s) plus all projection columns. This wider intermediate relation is then further clustered in any additional passes of the
Radix-Cluster algorithm. Partitioned Hash-Join on the matching clusters works as described just above.

While the algorithms do not change conceptually, the performance of a relational engine will be affected by increased CPU and memory access cost:

- as the input relations for Hash-Join or Radix-Cluster are not BATs, but full relations consisting of tuples that span all columns, scanning them causes (much) more memory access, which in case of queries that only project on a minority of the columns, will lead to inefficient cache-line usage. This point was also made in Section 6.3.2.

- relational operator implementations must manipulate tuples in a generic way, and cannot use the Monet code-expansion techniques described in the previous section (which generates a different routine at compile-time for each data-type processed). The reason why this is impossible is that there are too many possible combinations of tuple constellations to generate routines for (the number of key columns and projection columns, and their types cause a combinatorial explosion). As a consequence, an interpreter mechanism must be used in a relational engine, that examines each tuple and calls a type-specific method/routine for processing it, which typically leads to at least one hard-to-predict CPU branch (function call) for each value in a tuple. This again leads to mis-prediction stalls and memory stalls (due to lack of parallel memory access). On various kinds of query loads in relational DBMS technology, low levels of Instruction Level Parallelism (ILP) have been demonstrated – often barely one instruction per cycle, while modern CPUs are capable of 5 or more – due to such stalls, which themselves often account for 30%-50% of execution time alone [ADHW99, KPH+98, BGB98, TLPZT97].

In order to measure the performance of "traditional" (Partitioned) Hash-Join in a relational context while using Monet as our experimentation vehicle, we created the new integerX Monet atomic types in an extension module, for all $X \in \{1,4,16,64,256\}$. An integerX value models a relational tuple that stores $X$ simple integer column values. The implementation of the atom ADT routines for this type try to mimic the behavior of a multi-column relational engine; e.g., copying an integer8 during Partitioned Hash-Join or Radix-Cluster involves copying 8 integers with memcpy from soft-coded record offsets. The reason for this is that at DBMS kernel compile time, a relational engine can make no hard assumptions on the table formats that pass through the algebraic operators. Therefore, value handling is more interpretative and at least involves separate ADT calls for each column value. In the case of Monet, the fact that each operator works on a fixed table type (i.e. 2-column tables) enables it to hard-compile many optimizations such that typically the inner loop of a join algorithms does not contain a single method call (like described in Figures 6.11 and 6.18).

For experimentation convenience, we introduce in the same extension module a conversion command

\[
\text{[integer]}(\text{BAT(any, integerX)}, \text{int Y}) : \text{BAT[any, integerY]}
\]

that constructs a new BAT with a thinner or wider integerX column, either by omitting values, or by padding them. In the case of thinning (i.e. \(Y<X\)) a read-only BAT,
6.5. JOIN PROCESSING WITH PROJECTIONS

![Diagram of relational experiment in MIL]

Figure 6.23: Relational Partitioned Hash-Join Strategy

the command optimizes performance by constructing a view on the original BAT at zero cost (see Section 5.3.2).

The basic non-partitioned relational Hash-Join strategy would be coded in MIL like this:

01 smaller_all := [integer]([integer](smaller_key,1).reverse, 128).reverse;
02 larger_all := [integer]([integer](larger_key,1).reverse, 128).reverse;
03 smaller_project := [integer](smaller_all.reverse, 8).reverse;
04 larger_project := [integer](larger_all.reverse, 8).reverse;
05 res_join := join(larger_project, smaller_project.reverse);

In the example case above, we use “smaller” and “larger” tables that each have 128 columns, and the projection widths are $Y=Z=8$; we emulate relational storage of both tables in Monet by storing them as BAT[integer128,integer1], where the tail columns contain the “key” value and the head contains all other columns.

Notice that the fact that smaller_project and larger_project are MIL views on the base BATs smaller_all and larger_all, means that the projection is not materialized. Projection only occurs implicitly when the join in line 05 accesses both views – just like what would happen in a “traditional” relational system performing projection. Furthermore, the copying of each integer8 (view) value from its storage as integer128
is done with 8 `memcpy` calls that fetch values at regular intervals (i.e. at positions 0, 16, 32, 48, 64, 80, 96 and 112). In practice, this means that each `memcpy` causes a memory cache miss.

We can encode the Partitioned Hash-Join in the relational case by modifying the latter part of the script:

```sql
... 05 cluster_smaller := radix_cluster(smaller_project, P, H);
06 cluster_larger := radix_cluster(larger_project, P, H);
07 res_join := phash_join(cluster_larger, cluster_smaller.reverse, H);
```

Here we use primitives that are introduced in the `radix` extension module, that introduce the Radix-Cluster and Partitioned Hash-Join algorithms as new MIL operators. The first parameter of `radix_cluster` is a BAT that is to be clustered on its head column, its output is a materialized result BAT, and the other parameters are the number of passes and the number of Radix-Bits. As mentioned before, the number of Radix-Bits are divided equally over all passes (with a maximum difference of one due to the Radix-Bits not being an exact multiple of the number of passes). The `phash_join` is the implementation of the Partitioned Hash-Join that assumes inputs with the join columns clustered in H Radix-Bits, which is the additional third parameter.

This relational strategy is expected to have a different performance characteristic than the join experiments described in the previous Section, which did not take projection columns into account. First, there will be many more cache misses due to the reasons described above. Even Radix-Cluster cannot avoid those cache misses, as it is inherent to the relational storage format of base data. Second, there will be much more CPU cost, due to the fact that function-call overhead cannot be eliminated. In Monet algorithms, CPU optimizations cause a four-fold performance improvement, as described in Section 6.4.2. Third, the fact that the projection columns are taken as “extra luggage” with the join keys through the Partitioned Hash-Join means that the tuple size becomes larger: instead of clusters with `[key,oid]`, we have `[key,oid,integerX]` tuples in the clusters. As the clusters are tuned to fit into the cache, the clusters can hold less tuples, so we need to create more (and smaller) clusters in the Radix-Cluster phase. This may even cause the Radix-Cluster to make additional passes, and certainly will increase its cost, as we always have observed that to be monotonically increasing with the amount of Radix-Bits.

We evaluate the performance of this strategy in Section 6.5.3.

### 6.5.2 Cache-Conscious Join: Monet Strategies

As for join processing in Monet with column projections, the situation is also more complex than described in Section 6.4, though for different reasons than in a “traditional” relational query engine. In the case of Monet, a join query is processed by first constructing a join-index BAT`[oid,oid]`. This join-index is marked on both sides to form to “pivot” BAT`[oid,oid]` with a newly numbered head column (a dense sequence 0..`N-1`) and in the tail column the tuple IDs (`oid-s`) from either input relation. Subsequently, the projections are materialized by joining the pivot of the input relation with each projection column-BAT`[oid,T]`. This is depicted in Figure 6.24.

This strategy would yield the following MIL statements for our basic join-project SQL query:

```sql
... 01
```
6.5. JOIN PROCESSING WITH PROJECTIONS

Figure 6.24: Standard Monet Partitioned Hash-Join Strategy
# the physical join algorithm is either positional-, merge- or Hash-Join.
res_join := join(larger_key, smaller_key.reverse);
res_larger := res_join.mark.reverse;
res_smaller := res_join.reverse.mark.reverse;

## positional-join projected columns from smaller table into result
res_a1 := join(res_smaller, smaller_a1);
res_aY := join(res_smaller, smaller_aY);

## positional-join projected columns from larger table into result
res_b1 := join(res_larger, larger_b1);
res_bZ := join(res_larger, larger_bZ);

The join-index is named res_join, the pivots res_larger and res_smaller, the projection columns smaller_a and larger_b.

A Positional-Join is a highly efficient kind of join found in the Monet system, that occurs when an oid-column is joined with a void column. It is easy to lookup a value in a void-column, as the value you look up already tells its position (a search accelerator like hash-table or B-tree is not necessary, and CPU-cost is very low). The Positional-Join algorithm joins an outer BAT[any,oid] with an inner BAT(void,any) by scanning over the outer-BAT and performing positional lookup into the inner BAT.

In a typical data warehouse, the join at line 03 would be positional if the "key" columns are foreign keys between tables with a 1-1, 1-N or N-1 relationship (in those cases, one of the key columns would be of type void). However, if the columns are a N-M relationship, or if they do not form a foreign key at all, we could use the Partitioned Hash-Join as described earlier in this chapter, as this is a generic join algorithm that optimizes use of memory- and CPU-resources.

The Partitioned Hash-Join uses Radix-Cluster to quickly cluster both the smaller_key and larger_key BATs into clusters that fit the memory cache, and then repeatedly performs Hash-Join on the corresponding clusters. So instead of the simple MIL script above, we use the following alternative MIL statements to generate res_join:

00 # first radix cluster both key columns on H bits in P passes
01 cluster_larger := radix_cluster(larger_key, P, H);
02 cluster_smaller := radix_cluster(smaller_key, P, H);

# partitioned hash join on clusters of H Radix-Bits.
03 res_join := phash_join(cluster_larger, cluster_smaller.reverse, H);

The latter phase of the query (lines A1-AY,B1-BZ) fetches column values from the projected columns using Positional-Join.

BX res_bx := join(res_larger, larger_bx);

This performs fine with increasing table sizes of the larger table up until one larger_b column-BAT exceeds the size of the memory cache.

We now turn our attention to what happens if this happens. First, we discuss what happens if a larger_b BAT\(^9\) does not fit the memory cache. Then, we discuss what happens if even a smaller_a BAT plus its hash-table does not fit anymore.

\(^9\)These column-BATs may differ in tail-type and hence byte-width, but for simplicity we assume all have approximately the same size.
6.5. JOIN PROCESSING WITH PROJECTIONS

Ordering The Join Index to Improve Memory Access of Projections

If the BATs storing the columns of the “larger” table do not fit the memory cache anymore, the Positional-Joins in the last X statements of the MIL script will start to generate cache misses. This is caused by the fact that the oid-s in the tail of the res_larger BAT[void,oid]-s are not sorted; hence the access to the larger_b column-BATs is random.

The sorted projection strategy solves this problem, by sorting the result of the join first on the oid-s that point to the “larger” table (the head column of res_join):

```java
03 res_JOIN := join(larger_key, smaller_key.reverse);
res_JOIN_ordered := res_JOIN.reverse.order.reverse;
04 res_larger_reordered := res_JOIN_reordered.mark.reverse;
05 res_smaller := res_JOIN_reordered.reverse.mark.reverse;
...  # positional-join projected columns from larger table into result
B1 res_bl := join(res_larger_reordered, larger_bl);
EX ....
B2 res_b2 := join(res_larger_reordered, larger_b2);
```

As a result, the res_larger_reordered will be a BAT[oid,oid] ordered on tail, hence the Positional-Joins on the larger_b columns will cause a nice sequential access to both res_larger (as it is scanned in its role as “outer” join operand) and larger_b. We must, however, take into account that res_JOIN_reordered may be a BAT[oid,oid] that itself is larger than the memory cache, in which case the sorting operation itself could have caused a great many cache misses itself, and therefore perform badly. Let us therefore shortly discuss the memory access properties of various sorting algorithms.

Monet uses a CPU-optimized Quick-Sort algorithm for sorting large relations. The CPU-optimizations reduce the amount of function calls, by doing all value-comparison
and data movement inline, using C macros. In this sense it differs from the standard unix library call qsort, as that routine compares values with a user-provided function, and (often) moves values with memcpy (see also Section 5.13).

The memory access pattern of the Monet Quick-Sort consists of one sequential scan per recursion level (walking two cursors simultaneously, one from the start of the BAT forward, as well as another from the end of the BAT backward, until both meet in the middle). This is depicted in Figure 6.26. Quick-Sort is binary recursive and therefore takes \( \log_2(\text{ntuples}) \) recursion levels to sort a BAT, hence its total memory access consists of \( \log_2(\text{ntuples}) \) sequential scans. However, since Quick-Sort zooms into ever smaller sub-chunks of the BAT, there will be cache re-use in the deeper recursion levels as soon as such a chunk fits the memory cache, which happens when \( \text{sizeof(chunk)} = \text{sizeof(BAT)/(2^{level})} \leq \text{sizeof(cache)} \). Hence, the total memory cost of Quick-Sort is \( \log_2(\text{ntuples}) - \log_2(\text{sizeof(cache)/sizeof(tuple)}) \) sequential scans.

This \( O(N \log(N)) \) complexity – with rather low \( \log(N) \) – means that the Monet Quick-Sort implementation behaves quite good both concerning CPU efficiency and memory access pattern. Still, for some simple data types, in particular columns containing oid-s, one can further improve the memory access performance by using Radix-Sort instead of Quick-Sort.

Radix-Sort is essentially a Radix-Cluster on all bits, hence we do:

```java
03 res_join := join(larger_key, smaller_key.reverse);
   res_join_ordered := res_join.reverse.radix_cluster(P1, H1).reverse;
```

Where \( P_i \) is a suitable number of Radix-Cluster passes and \( H_1 \) is here the total number of “significant bits”, where we define the most significant bit in a collection of cardinal integer values as the highest bit set in the binary representation of its largest value. The head column of the \( \text{join(larger_key, smaller_key.reverse)} \) is of type oid, and contains the oid-s from the matching tuples in the “larger” table. Table-oid-s are automatically generated by the void columns of Monet, and therefore these integer values are from the range \([0, ..., N-1]\), where \( N \) is the number of tuples in the “larger” table. We call such an integer sub-domain a “dense” domain. We hence see that in
dense domains, the number of significant bits is minimal (i.e. $H_l = \log_2(N)$ – there are no “spoiled” values), and we do not expect skew in such a column. This motivates our choice to implement Radix-Cluster for the $\text{oid}$ type by getting Radix-Bits without hashing (for all other types, we hash first). Hashing is not necessary due to absence of value-skew on $\text{oid}$ columns, and absence of hashing allows us to use Radix-Cluster as Radix-Sort.

Going one step further, the partial-cluster projection join strategy supplants Radix-Cluster on all significant bits (i.e., Radix-Sort), by a Radix-Cluster on less bits. For this purpose, we added the possibility to indicate to the Radix-Cluster to ignore a certain number of lower bits (by passing the number of Radix-Bits to ignore as an extra last parameter to $\text{radix\_cluster}$). This in fact breaks off Radix-Sort, leaving the relation unsorted on the lowermost bits. The relation is sorted, though, on the highermost bits (i.e. partially ordered). This partial ordering means that in each cluster all values fall in a certain disjunct range. When entering a Positional-Join, this means that each cluster will only fetch join values in the column-BAT in a certain range (or cluster). If these “virtual” clusters in the column-BAT[void,T] fit in the memory cache, then the Positional-Join will run well (i.e., not thrash the cache). Note that the maximum size for which this is the case also depends on the byte-width of type $T$. The benefit of this “partial-cluster” join strategy (depicted in Figure 6.27) is that it has the potential to optimize memory performance of the column fetching using Positional-Joins just as well as a full sort, but at a clustering cost that is less than the cost of a full sort.

The detailed performance evaluation for using Quick-Sort or (full) Radix-Cluster is found in Section 6.5.3.
The Radix-Decluster Algorithm and its Uses

Let us now return to the original motivation for discussing sorting, which was to sort the join result (join-index) on the oid-s of the "larger" relation, if its column-BAT[void,T]-s were too large to fit the cache (which would result in bad memory performance with an unsorted pivot, due to the random access pattern). However, in a generic project-join query, there are not only projections to be done from the "larger" table, but also from the "smaller". Hence, if the column-BAT[void,T]-s of the "smaller" table are also larger than the memory cache, the same problem occurs for the positional joins to these column-BATs. It is clear that the join-index cannot simultaneously be sorted on oid-s of the "larger" table and on those of the "smaller" table. Hence, the sorting of the join-index on the "larger" table must still be done first. One could possibly re-order the pivot for the "smaller" table (which is created by mark on the join-index) afterwards:

```
05 res_smaller_sorted := res_join.reverse.mark.reverse.radix_cluster(Ps,Hs);
```

However, this approach only transfers the problem to later phases of query processing, as follows. The Positional-Joins of res_smaller_sorted into the smaller_a column-
BAT[void, T]-s would run fine, but as a tail-sorted res_smaller_sorted would turn into a BAT[oid, oid] (i.e. it would no longer have a void head column), the result of these Positional-Joins would be of the form BAT[oid, T]. These results would not only take more space than the desired form BAT[void, T], but would also create a problem in further use of the query result, as these res_{a,y} BATs will not be sorted on head. Join access to them would go to the Hash-Join rather than the Positional-Join, and due to the random access this would pose a memory caching problem as these res_{a,y} BATs may well be larger than the memory cache.

As a solution, we propose the use of a new memory-conscious join strategy for these projection joins, that is called cluster-decluster. This strategy, which is illustrated in Figure 6.28, consists of three phases:

1. Partial Radix-Cluster:

   First, the res_smaller is partially Radix-Clustered on tail-oid. That is, relation PIVOT is clustered on some number of Radix-Bits, but not on all (significant) Radix-Bits. The number of bits chosen for the practical ordering is the minimal number of bits such that in a Positional-Join the oid-s in each partially sorted cluster point to a memory range in COLUMN that fits the memory cache. For example, if we have a memory cache of 64KB and we assume values to be 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 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 $\log_2(10M) = 24$).

2. Clustered Positional-Join:

   The purpose of the Radix-Clustering PIVOT in the previous phase is to accelerate the Positional-Join between PIVOT and COLUMN (e.g. res_smaller and smaller_{a,y}). Because the oid-s in the tail of PIVOT are now partially sorted, each chunk in PIVOT will only randomly access data from one “cluster” in COLUMN. As we have chosen the partial ordering bits in such a way that this randomly accessed region fits the memory cache, during Positional-Join these regions in COLUMN stay memory resident, and Positional-Join will cause much less cache misses than with an unclustered PIVOT. The result of joining the clustered PIVOT with COLUMN is a BAT[oid, T]. The head type is not void because the head type of the clustered PIVOT is not either. One can additionally optimize performance by splitting the clustered PIVOT by column in two BATs, using a mark on both sides of PIVOT, yielding one BAT[void, oid] called CLUST_OUTPUTIDS, that contains the oid-s of the join result table in the tail, and another BAT[void, oid] called CLUST_INPUTIDS, that contains the oid-s of the smaller (input) table in the tail. This latter BAT is then used to perform the positional join into COLUMN. The advantage here is that the resulting BAT[void, T] - which we call CLUST_VALUES - has a void head column. Recall that this join has the special property that each tuple hits exactly once, and that the result tuples appear exactly in the order of the left join operand, which enables the positional join implementation to produce a result with a void head column if its left operand has one.

3. Radix-Decluster:
In order to complete the operation, we now want to produce the tail values of CLUST.VALUES in the order dictated by the tail column of CLUST.OUTPUTIDS. We also know that the tail values of CLUST.OUTPUTIDS would form a dense sequence \((0, 1, \ldots, N-1)\) when sorted (as this column was the result of re-clustering the dense head column of PIVOT). This "declustering" can be done efficiently by exploiting the property of the Radix-Cluster algorithm: following a Radix-Cluster on the tail of a BAT that is ordered on head, each cluster in the result will have the head-values still ordered within the cluster. So, the tail values within each cluster in CLUST.OUTPUTIDS appear in order. Due to the fact that the clusters are sorted, a full sort can be efficiently implemented as a merge. This merge operation processes all clusters of \([\text{oid}, T]\) tuples, holding output-ids and column values, that are stored in the two BATs CLUST.OUTPUTIDS and CLUST.VALUES. The result of merging all tuples on output-ids order is a BAT\([\text{void}, T]\) because, when sorted, the output-ids form a dense sequence. Normally, the cost of a merge of \(N\) tuples partitioned over \(H\) sorted clusters is at least \(O(\log_2(H)N)\). However, because of the fact that we know beforehand that the sorted sequence becomes dense, we can bring this cost back to \(O(N)\)!

The Radix-Decluster algorithm, depicted in detail in Figure 6.29, works by keeping open an insertion-window of \(s\) oid-s \([w, w+1, \ldots, w+s-1]\) during the merge.
Each iteration of the algorithm finds all the next s oid-s in the clusters and appends them to the result BAT[void,T]. This is done by going through all (not yet empty) clusters and inserting from the top of each cluster all elements whose head oid fits in the window. Due to the fact that all clusters are sorted on head, we are sure to have found all oid-s after having processed all clusters once. Then, the window is shifted s positions and the process repeats. The window size s is preferably much larger than the number of clusters, such that per iteration in each cluster multiple tuples fall into this window. Because these multiple tuples are accessed sequentially in both CLUST_OUTPUTIDS and CLUST_VALUES from the top of the cluster downward, the cache lines that store these BATs are used fully; giving efficient memory performance. The only restriction on s is that the window size in the result BAT[void,T] must fit the memory cache, as it is accessed randomly. Pseudo code of the algorithm is in Figure 6.30 while Figure 6.29 gives a detailed example of how it works.

The cluster-decluster strategy is expressed in the following MIL code:

```mil
A0 # positional-join and decluster projected columns from smaller table into result
   clust_smaller_outputids := res_smaller_clustered.radix_count(Hs, Ri);
   clust_smaller_outputids := res_smaller_clustered.mark.reverse;
A1 res_a1 := join(clust_smaller_inputids, smaller_al);
   radix_decluster(clust_smaller_outputids, clust_smaller_borders);
A2 ....
A3 res_an := join(clust_smaller_inputids, smaller_an);
   radix_decluster(clust_smaller_outputids, clust_smaller_borders);

The radix_count operator analyzes a (partially) Radix-Clustered BAT and returns the actual sizes of the clusters. It returns a BAT[void,int] where for each bit pattern, the tail contains the size of the cluster. These sizes are used in the Radix-Decluster to initialize the cluster border structure.

The cluster-decluster join strategy is more expensive than the partial-cluster join strategy. Both strategies feature one initial Radix-Cluster, 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, if cache-conscious projecting is already needed for the join input table with the more expensive projections. Which input table in the join has the more expensive projection phase depends on the number of projection columns in both tables, the data types in these projection columns, and the number of tuples in both input tables. In the MIL script for cache conscious query processing given below, we use partial Radix-Cluster for getting the columns from the “larger” table, and the cluster-decluster strategy for getting the projection columns from the “smaller” table:
```

```mil
# first radix cluster both key columns on H bits
01 cluster_larger := radix_cluster(larger_key, L1, ..., Ll);
02 cluster_smaller := radix_cluster(smaller_key, S1, ..., Ss);

# phash join on clusters of H Radix-Bits, followed by Radix-Sort on head column.
03 res_join := phash_join(cluster_larger, cluster_smaller.reverse, H);
   res_join.reverse.radix_cluster(R1, ..., Rp).reverse;
```
CHAPTER 6. MEMORY/CPU OPTIMIZED QUERY PROCESSING

\begin{verbatim}
<Type>{
radix_decluster<Type>(
  int cardinality, nclusters,
  Type value_bat[cardinality],
  oid id_bat[cardinality],
  struct { int start, end } cluster[nclusters])}{
  <Type> result_bat[] = 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 (id_bat[cluster[i].start] < windowLimit) {
        result_bat[id_bat[cluster[i].start]] = value_bat[cluster[i].start];
        if (++cluster[i].start >= cluster[i].end) {
          cluster[i] = cluster[-nclusters]; // delete empty cluster
          if (i >= nclusters) break;
        }
      }
    }
  }
  return result_bat;
}

Figure 6.30: The Radix-Decluster Algorithm in Pseudo C++

04 res_larger_sorted := res_join.mark.reverse;
# sub-cluster on Rs significant Radix-Bits, ignoring lowest Ri bits
05 res_smaller_cluster := res_join.reverse.mark.reverse.radix_cluster(-Ri, Rs);

# positional-join and decluster projected columns from smaller table into result
08 borders := res_smaller_clustered.radix_count(Rs, Hi);
A1 res_a1 := join(res_smaller_clustered, smaller_a1).radix.decluster(borders);
AX ...
AY res_aY := join(res_smaller_clustered, smaller_aY).radix.decluster(borders);

# positional-join projected columns from larger table into result
B1 res_b1 := join(res_larger_sorted, larger_b1);
BX ...
BZ res_bZ := join(res_larger_sorted, larger_bZ);

Radix Accelerator

First we show the overloaded join MIL procedure that integrates the radix-cluster and partitioned hash-join algorithm seamlessly in the MIL equi-join using some dynamic tactical query optimization: depending on the actual operand sizes and on the global CACHE_SIZE and CACHE_LINES variables – which are initialized by the Calibrator module at database startup to the size of the performance-wise most significant memory cache level and number of cache lines – it determines whether or not to use partitioned hash-join, which operand is the inner and outer and the optimal Radix-Bits and passes.

PROC join(BAT[void,any::1] left, BAT[any::1,any::2] right) : BAT[oid,any:2] {  
  VAR right_batsize := right.info.find("batsize");
  VAR left_batsize := left.info.find("batsize");

  IF (min(right_batsize,left_batsize) > CACHE_SIZE) {  
    IF (left_batsize > right_batsize) {  
      var nbits := 1 + log2(left_batsize/CACHE_SIZE);
      var npasses := 1 + (nbits-1) / CACHE_LINES;
      RETURN phash_join(right.radix_cluster(npasses, nbits).reverse,
                         left.reverse.radix_cluster(npasses, nbits), nbits).reverse;
    } ELSE {  
      var nbits := 1 + log2(right_batsize/CACHE_SIZE);
      var npasses := 1 + (nbits-1) / CACHE_LINES;
    }
  }
}
\end{verbatim}
6.5. JOIN PROCESSING WITH PROJECTIONS

In order to automatically apply the cluster-decluster join strategy in MIL queries that can benefit from it, the algorithms have also been wrapped in a MIL **radix search-accelerator**. As discussed in Section 4.4.2, such structures are attached to BATs, may be maintained under updates and can be exploited by algebraic operators to accelerate their execution. While the Monet features built-in direct hashing and T-trees, new search accelerators can be added in extensibility modules.

The **radix.cluster**(X, B) operator attaches its clustered result Y to the input BAT[void, oid] X and in turn attaches **radix.count**(Y, B) to Y by use of a new MIL search accelerator. Also, the **radix.cluster**(X) directly returns such an attached Y if found on an X input. The same goes for **radix.count**(Y), when it discovers an attached borders BAT on a clustered input Y.

The second overloaded join MIL procedure below makes sure that the cluster-decluster join strategy is applied in Positional-Joins (i.e., those with a void head column in the right join operand) whenever the input column is not sorted or clustered, and the target column is larger than the memory cache size.

```plaintext
PROC join(BAT[void,oid] left, BAT[void,any::1] right) : BAT[void,any::1] {
    VAR right_info := right.info();
    VAR right_size := int(right_info.find("batsize"));
    if (int(right_info.find("tail.sorted")) * 0 and right.batsize > CACHE_SIZE) {
        VAR nbits := 1 + log2(right.batsize/CACHE_SIZE);
        VAR nignore := min(0, (1 + log2(left.count)) - nbits);
        VAR npasses := 1 + (nbits-1) / CACHE_LINES;
        VAR cluster := left.radix_cluster(npasses, nbits, nignore); # only computed first time
        VAR borders := cluster.radix_count(nbits, nignore); # only computed first time
        VAR cluster_values := cluster.reverse.mark.reverse;
        VAR cluster_ids := cluster.mark.reverse;
        RETURN cluster_values.join(right).radix_decluster(cluster_ids, borders);
    }
    return join(left, right);
}
```

### 6.5.3 Performance Evaluation

In this section, we present experiments done on a single-CPU node of the Origin2000 described in Table 6.1. In these experiments, we executed our example project-join SQL query with the above described Monet and "traditional" relational query processing strategies on relations of equal size \( C \in \{125K, 500K, 2M, 8M\} \), consisting of \( W \in \{1, 4, 16, 64\} \) all-integer (4-byte) columns, with a join hit rate of 3, and projecting \( P \in \{1, 4, 16, 64\} | P \leq W \) columns from both relations into the result. In all experiments, all processing happens in main-memory (no I/O or page faults).

It should be noted, that in all our experiments, apart from elapsed time, we also measured memory cache misses in both the L1 and L2, as well as branch mis-predictions. In the evaluation of join query performance with projections, we omit the detailed measurement results as well as performance models, similar to those presented earlier in this chapter for the Radix-Cluster and Partitioned Hash-Join algorithms. We do this for reasons of conserving space, and also because performance behaves as expected:
CHAPTER 6. MEMORY/CPU OPTIMIZED QUERY PROCESSING

<table>
<thead>
<tr>
<th>cardinality C</th>
<th>reorder</th>
<th>posjoin</th>
</tr>
</thead>
<tbody>
<tr>
<td>125K</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>500K</td>
<td>0</td>
<td>215</td>
</tr>
<tr>
<td>2M</td>
<td>0</td>
<td>2853</td>
</tr>
<tr>
<td>8M</td>
<td>0</td>
<td>15928</td>
</tr>
<tr>
<td>unsorted</td>
<td>347</td>
<td>29</td>
</tr>
<tr>
<td>quick-sort</td>
<td>1620</td>
<td>118</td>
</tr>
<tr>
<td>radix-cluster</td>
<td>7456</td>
<td>484</td>
</tr>
<tr>
<td>4KB</td>
<td>3474</td>
<td>33</td>
</tr>
<tr>
<td>16KB</td>
<td>1592</td>
<td>88</td>
</tr>
<tr>
<td>64KB</td>
<td>2362</td>
<td>22</td>
</tr>
<tr>
<td>256KB</td>
<td>19460</td>
<td>23</td>
</tr>
<tr>
<td>1MB</td>
<td>19713</td>
<td>22</td>
</tr>
<tr>
<td>4MB</td>
<td>15779</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 6.3: Positional-Join(3C, C) Performance (ms) without reordering, with Quick-Sort(3C) or Radix-Cluster(3C) (possibly +Radix-Decluster(3C))

- Quick-Sort indeed exhibits $C(\log_2(C) - \log_2(\text{cachesize}))$ cache misses on all memory cache levels,
- Unsorted Positional-Join into a column that is larger than a cache, thrashes that memory cache (degrading with increasing cluster size to the worst-case of one cache miss for each tuple in the outer relation).
- Sorted or Clustered Positional-Join into a column with clusters that fit the cache, exhibits just the amount of misses to read in the column once (i.e. the minimal “compulsory” misses).
- Radix-Decluster with an insertion window size smaller than the cache only exhibits the minimal compulsory misses to read in all input relations once.

Considering the Monet strategy, a first question of interest is which reordering algorithm works best: Quick-Sort or Radix-Sort. Figure 6.31 shows results for various re-ordering algorithms (Quick-Sort, Radix-Sort and Radix-Cluster) and for Positional-Join. The reordering done in our example setting is on the join index previously computed with Partitioned Hash-Join. This BAT[oid,oid] has a cardinality of 3C, due to the join hit-rate of 3. The join costs in this table are for joining this 3C join index into a column-BAT[void,int] of size C. The reordering costs are for reordering a 3C join index. The results show that Quick-Sort is consistently beaten by Radix-Sort on oid, which is achieved by Radix-Cluster on all significant bits ($B = \log_2(C)$).

The detailed data in in Table 6.3 show that in general, however, best performance is not obtained with Radix-Sort but with (partial) Radix-Cluster that clusters on a much more coarser cluster-size than 1 tuple (recall Radix-Sort is a Radix-Cluster on all bits). Partial Radix-Cluster makes Positional-Join performance deteriorate a bit, but at the benefit of greatly reduced reordering cost. The fewer Radix-Bits, the better the performance (see also the left graph in Figure 6.32). However, in order to make Positional-Join work well, the clusters should at least fit the L2 cache, and keep improving until they approximate the cost of Positional-Join on a sorted input, when
6.5. JOIN PROCESSING WITH PROJECTIONS

Figure 6.31: Performance of Sorting(3C) and Positional-Join(3C, C)

the clusters are so small that they fit the L1 cache. In the case of the Origin2000, a reasonable cluster size seems to be 256KB, which is one sixteenth of the L2 size, but still larger than the 32KB L1. For the larger cardinalities (e.g., 2M and 8M), we see that even if there is only one projection column, Radix-Cluster followed by Positional-Join beats Unsorted Positional-Join, and the performance gain can become a factor 6 if there are many projection columns.\(^\text{10}\)

The partial-cluster join strategy can only be used for projecting columns of one of the two join relations efficiently with respect to the memory caches. However, if we need to project columns from both relations, and both relations have so many tuples that their column-BATs exceed the cache size, we need to use the cluster-decluster strategy for doing the projections from the other table. The performance of the Radix-Decluster operation is depicted in the right graph of Figure 6.32. The performance of Radix-Decluster is quite flat, only deteriorating due to memory cache misses when the amount of clusters becomes too large (and the cluster sizes too small). We should note that we found performance of Radix-Decluster to be optimal in this configuration when the insertion window was kept to half of the L2 size (which means that there are still a substantial number of L1 misses).

Getting back again to the detailed numbers in Table 6.3, we should look at the numbers added to the join cost (listed after the plus sign); this is the cost of Radix-Decluster to create the final column result. In the cluster-decluster strategy, each Positional-Join is followed by a \texttt{radix\_decluster} operation; therefore their costs have to be added. This decreases the performance advantage with respect to unclustered Positional-Join, but one can see that for the larger cardinalities (2M and 8M), Radix-

\(^{10}\)With larger cardinalities and faster CPUs, the relative penalty of cache thrashing in the Unsorted Positional-Join becomes larger, so the overall gain will improve over time.
CHAPTER 6. MEMORY/CPU OPTIMIZED QUERY PROCESSING

Figure 6.32: Performance of Radix-Cluster(3C) and Radix-Decluster(3C)

Decluster still wins.

We now analyze which (cache-conscious) query processing strategy for the projection phase of our generic join query works best in Monet and under which circumstances. As described earlier, we consider tables of up to 256 columns, which is the maximum value of $P$ here. Note that for Monet query performance only $P$ matters, not the actual number of columns in the table $W$ (as they are fragmented vertically in distinct column-BATs). Therefore, a Monet experiment for a certain $P$ holds for all $W$.

We consider four strategies:

u **Unsorted**: one Positional-Join from the join index into each projected column-BAT.

s **Sorted**: first sort the join index with the best sorting algorithms (i.e. Radix-Sort).

p **partial-Cluster**: first partially cluster the join index. We take the number of Radix-Bits that gives the best result (in this setting, this generally boils down to the 256KB cluster size).

d **cluster-Decluster**: like the clustered experiment, but each Positional-Join is followed by Radix-Decluster.

Figure 6.33 summarizes the performance of the various Monet strategies to process the projection phase of our example SQL join, depending on amount of projection columns $P$ and cardinality $C$ (recall that the join result actually has size $3C$ due to the join hit ratio of 3). For small cardinalities ($C \leq 125K$), all strategies that do any kind of reordering lose to simple unsorted processing of the Positional-Joins, since the column-BATs 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 more than a factor
6.5. JOIN PROCESSING WITH PROJECTIONS

![Graph showing projection strategies comparison](image)

**Figure 6.33:** Monet Projection Strategies: Unsorted vs Sorted vs Radix-Cluster vs Radix-Decluster

10 at $C = 32M$ and $P = 64$). Like we said before, partial-clustered is always more efficient than sorted processing; the difference becoming larger with more projection columns. Finally, we see that the cluster-decluster strategy always loses from the partial-cluster strategy, but is actually quite competitive, beating unsorted processing by a large margin. One should therefore choose it if one cannot use partial-cluster anymore (i.e. on the second projection table).

As such, we formulate the following rules to arrive at the “optimal” projection strategy in Monet on the Origin2000:\footnote{The rules formulated here are specific to the characteristics of the Origin2000, which has a slow TLB and fast L1 cache. Calibrated hardware characteristics for use in detailed cost models, like those formulated earlier in the Chapter for Radix-Cluster, will provide full performance insight that enables a query optimizer to choose the optimal strategy for any query on any hardware. The development of such detailed performance models for Radix-Decluster and Positional-Join is considered future work here – though it should not be difficult given the consistent performance results we encountered.}

1. if for all relations the largest column-BAT fits the L2 cache, use unsorted processing.

2. if there is only one relation that has projection columns and that does not fit the above rule, use the partial-cluster strategy for the projections from that relation, and use the unsorted strategy for the projections from the other relations.
### Table 6.4: Join-Project Performance (ms): Monet vs Relational, simple hash-join → fully cache optimized

<table>
<thead>
<tr>
<th>$P$</th>
<th>system</th>
<th>$W$</th>
<th>$C=125K$</th>
<th>$C=500K$</th>
<th>$C=2M$</th>
<th>$C=8M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>monet</td>
<td>*</td>
<td>168 (u/u)</td>
<td>932 (u/u)</td>
<td>4871 (c/d)</td>
<td>33069 (c/d)</td>
</tr>
<tr>
<td></td>
<td>relational</td>
<td>1</td>
<td>419 → 551</td>
<td>3033 → 2229</td>
<td>14892 → 8833</td>
<td>33069 (c/d)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>560 → 571</td>
<td>5275 → 2275</td>
<td>16590 → 8965</td>
<td>74948 → 47223</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>724 → 577</td>
<td>4827 → 2385</td>
<td>18191 → 9919</td>
<td>146674 → 48127</td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>965 → 613</td>
<td>4735 → 2635</td>
<td>30088 → 9911</td>
<td>161291 → 51477</td>
</tr>
<tr>
<td>4</td>
<td>monet</td>
<td>*</td>
<td>372 (u/u)</td>
<td>2063 (c/u)</td>
<td>9641 (c/d)</td>
<td>42502 (c/d)</td>
</tr>
<tr>
<td></td>
<td>relational</td>
<td>4</td>
<td>1011 → 1028</td>
<td>5928 → 4459</td>
<td>22882 → 12221</td>
<td>102837 → 81320</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>1349 → 1060</td>
<td>6602 → 4571</td>
<td>25790 → 14613</td>
<td>169883 → 87721</td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>1395 → 1160</td>
<td>6970 → 5767</td>
<td>46439 → 19611</td>
<td>248099 → 94887</td>
</tr>
<tr>
<td>16</td>
<td>monet</td>
<td>*</td>
<td>1133 (c/u)</td>
<td>5982 (c/u)</td>
<td>28509 (c/d)</td>
<td>161990 (c/d)</td>
</tr>
<tr>
<td></td>
<td>relational</td>
<td>16</td>
<td>2753 → 2955</td>
<td>12804 → 12580</td>
<td>53433 → 55849</td>
<td>161990 (c/d)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>64</td>
<td>3309 → 2989</td>
<td>17098 → 13257</td>
<td>77425 → 58235</td>
<td>161990 (c/d)</td>
</tr>
<tr>
<td>64</td>
<td>monet</td>
<td>*</td>
<td>3960 (c/u)</td>
<td>21294 (c/u)</td>
<td>100461 (c/d)</td>
<td>559464 (c/d)</td>
</tr>
<tr>
<td></td>
<td>relational</td>
<td>64</td>
<td>9992 → 11155</td>
<td>38441 → 46728</td>
<td>100461 (c/d)</td>
<td>559464 (c/d)</td>
</tr>
</tbody>
</table>

3. if both relations are big (i.e., their column-BATs do not fit L2) and both have projection columns, use partial-cluster on one and cluster-decluster on the other. One could use the heuristic to use partial-cluster on the relation with most projection columns, or with the smallest product of cardinality and number of projection columns. Notice that in our experimentation setting, where we have equal cardinalities and equal number of projection columns in both join relations, it does not matter which relation is processed with the partial-cluster strategy and which with the cluster-decluster strategy (this setting with equally difficult projections from both relations is actually worst-case for Monet, as we get maximal cluster-decluster cost).

In Table 6.4 we use the “optimal” strategy for Monet, and compare its performance (in milliseconds) with the performance of “traditional” relational query processing, both without and with Partitioned Hash-Join (displayed as without → with). For the Monet results, we append between parentheses the chosen strategy for projecting columns (displayed as larger/smaller).

Considering the optimal overall Monet strategy, we see that in the smaller sizes the $u/u$ – both unsorted – strategy wins. At some point, this transitions into $c/u$ (here, the declustered strategy is still more expensive than unsorted). At large cardinalities though, the unsorted strategy costs explode due to cache thrashing, and we get $c/d$ as a winner. We conclude that the Radix-Decluster operation is instrumental in providing an efficient generic join processing strategy in Monet.

One conclusion to be drawn is that also for relational DBMSs, Partitioned Hash-Join brings substantial advantages. In other words, the results of Section 6.4 not only hold for Monet, where we did our initial experiments, but also for generic RDBMS technology. We observe that the advantage is especially pronounced if the relation sizes are large and the percentage of columns being projected on is low. In those cases, a 3-fold performance improvement can be obtained (e.g., $C=8M, W=64, P=1$).

Another conclusion to be drawn is that vertical fragmentation as implemented in Monet beats “traditional” query processing by a margin that also is maximized at large cardinalities and low projection percentages, and can reach almost an order of magnitude (e.g., more than a factor 6 at $C=2M, W=64, P=1$) at our relatively small
6.5. JOIN PROCESSING WITH PROJECTIONS

Table settings.\textsuperscript{12}

Even if we compare with cache-optimized relational processing, the Monet advantage is more than 50\% (factor 2-3). It is interesting to note that although it is definitively the case that a Monet query plan moves more data around than a relational one, the higher efficiency with which Monet can execute its operations more than compensates for that. There are two reasons why the vertical fragmentation in Monet is instrumental in this: first, it helps reducing cache misses as columns that are not in the projection never need to be scanned and brought into the caches. Secondly, the vertical fragmentation allows MIL to be what it is: an algebraic language with operators that have signatures with little degree of freedom (few parameters, all of them 2-column tables). This small degree of freedom in the algebra gives the operators much knowledge at DBMS kernel compile-time, enabling CPU-optimizations that are otherwise not possible.\textsuperscript{13}

We have shown in Section 6.4 that such CPU optimizations can attribute for a factor 4 of performance improvement on modern CPUs.

We can argue though, that the advantage of Monet over "traditional" relational query processing is in fact even larger, and will expand in the future, for the following reasons:

- The experimental setting with equally sized join relations and equal numbers of projection columns is worst-case for Monet; as it maximizes Radix-Decluster cost. As Radix-Decluster typically takes at most one third of the query time in our experiments, the gains of having more skewed numbers of projection columns are limited, however.

- The relational implementation tested here uses a Monet infrastructure which we suppose to be more CPU-efficient in a main-memory setting than that of the average RDBMS implementation.

- Hardware advances will worsen the memory access bottleneck and increase the benefit of the extreme CPU-optimizations applied in Monet. At the time of this writing, 2 years after the experiments, the fastest CPU is the Pentium 4 running at 2.2GHz. This is a factor 5 faster than the Pentium III tested here, while memory latency has not improved (it actually became 20\% slower in RDRAM). Furthermore, the Pentium 4 has an extremely long pipeline (doubled in length with respect to the Pentium III) making it extremely vulnerable to branch mispredictions (and increasing the benefit of CPU optimizations). This vulnerability is even more present in the Itanium line of server processors, which due to its architecture (EPIC) fully relies on compile-time optimizations. Also, this line of processors exhibits an increasing abundance of parallel hardware resources which is sure to go unused in non-optimized DBMS code.

\textsuperscript{12}We actually carried out experiments up to $C=64$M tuples and $W,P=256$ columns, but the shortcomings of our relational implementation – i.e. the byte-size of one BAT could not exceed 32-bit integer bounds – limited the relational experiments to those where $P \times C < 128M$.

\textsuperscript{13}One other way to increase the compile-time optimization possibilities is to compile in real-time, e.g. by parsing SQL queries, generating C/C++ – or even assembly – and compile this just-in-time into a dynamically loadable library, which is then loaded and called into for executing the query. Compilation time can be considerable, though, which might be contradictory to ad-hoc query processing as in OLAP, but also could be alleviated by query-plan caching. To our knowledge, no current RDBMS implementations use just-in-time machine code compilation, but we pose that given its increasing benefits on modern hardware this may be a sound technique for future database systems.
6.6 Conclusion

We have discussed trends in modern hardware, showing the importance of good usage of the various cache levels in current computer systems in order to limit access to main memory DRAM chips, which gets exponentially more expensive with the law of Moore. Also, modern super-scalar CPUs have become fragile with respect to efficiency, as they depend on speculative execution to fill their long pipelines and only deliver their advertised performance if program code has much locality and is highly predictable.

In this chapter, we investigate the effects of these phenomena on query processing. For sequential table access, Figure 6.3 shows that the key issue is to reduce the stride, which is achieved in Monet by vertical fragmentation and enhanced with enumeration types (which makes BUNs, and thus the stride, smaller). As for query processing operators with random access, such as hash-join, we have shown that un-cacheable memory access patterns can lead to almost a magnitude of performance degradation in large equi-joins. We refined partitioned hash-join with a new partitioning algorithm called Radix-Cluster, that prevents performance becoming dominated by memory latency (avoiding the memory access bottleneck).

Exhaustive equi-join experiments were conducted on modern SGI, Sun, and PC hardware, and we formulated detailed analytical cost models that explain why and when the Radix-Cluster/Partitioned Hash-Join algorithm makes optimal use of hierarchical memory systems found in modern computer hardware and very accurately predict performance on all three platforms. This modeling work represents an important improvement over previous work on main-memory cost models [LN96, WK90]. Rather than characterizing main-memory performance on the coarse level of a procedure call with “magical” cost factors obtained by profiling, our methodology mimics the memory access pattern of the algorithm to be modeled and then quantifies its cost by counting cache miss events and CPU cycles. We were helped in formulating these models through our usage of hardware event counters present in modern CPUs.

Furthermore, we contributed a hardware-calibrator that automatically detects the hardware characteristics of the memory hierarchy, and forms the crucial model parameters for tuning the settings of our Radix-Algorithms. Such a software module should be part of any cache-conscious DBMS, as it allows to tune cache-conscious query processing algorithms independent of the hardware. Monet incorporates the calibrator as an extension module that is run at startup, and we showed with the Radix-Accelerator, how the extracted parameters are used to perform dynamic cache-optimized query processing through tactical query optimization in MIL.

Finally, we extended our study from isolated hash-join execution to generic join query processing including projections. Here we compared “traditional” relational query processing (with and without Radix-Algorithms) with various column-oriented query processing strategies in Monet. Here, we contributed another algorithm called Radix-Decluster, which in combination with Partial Radix-Cluster and efficient Positional-Join enables fully cache-optimized query processing in Monet. In the overall experiments, we have shown that optimizing equi-joins using the Radix-Algorithms improves performance both in the relational and in the Monet case, while the Monet implementation is consistently significantly faster than the relational approach.