Breaking the Memory Wall in MonetDB

Article *in* Communications of the ACM · December 2008

DOI: 10.1145/1409360.1409380 · Source: DBLP

Peter A. Boncz
Centrum Wiskunde & Informatica
143 PUBLICATIONS **5,749** CITATIONS
SEE PROFILE

Martin Kersten
Centrum Wiskunde & Informatica
302 PUBLICATIONS **7,816** CITATIONS
SEE PROFILE

Stefan Manegold
Centrum Wiskunde & Informatica
125 PUBLICATIONS **3,400** CITATIONS
SEE PROFILE

DOI:10.1145/1409360.1409380

Breaking the Memory Wall in MonetDB

By Peter A. Boncz, Martin L. Kersten, and Stefan Manegold

Abstract

In the past decades, advances in speed of commodity CPUs have far outpaced advances in RAM latency. Main-memory access has therefore become a performance bottleneck for many computer applications; a phenomenon that is widely known as the "memory wall." In this paper, we report how research around the MonetDB database system has led to a redesign of database architecture in order to take advantage of modern hardware, and in particular to avoid hitting the memory wall. This encompasses (i) a redesign of the query execution model to better exploit pipelined CPU architectures and CPU instruction caches; (ii) the use of columnar rather than row-wise data storage to better exploit CPU data caches; (iii) the design of new *cache-conscious* query processing algorithms; and (iv) the design and automatic calibration of memory cost models to choose and tune these cache-conscious algorithms in the query optimizer.

1. INTRODUCTION

Database systems have become pervasive components in the information technology landscape, and this importance continues to drive an active database research community, both academic and industrial. Our focus here is on so-called *architecture-conscious* database research that studies the data management challenges and opportunities offered by advances in computer architecture. This area of research started receiving impetus 10 years ago[1,2] when it became clear that database technology was strongly affected by the emergence of the "memory wall"—the growing imbalance between CPU clock-speed and RAM latency.

Database technology, as still employed in the majority of today's commercial systems, was designed for hardware of the 1970–1980s and application characteristics that existed at the time. This translates into the assumption of disk I/O being the dominating performance factor, and an architecture tuned to supporting so-called online transaction processing (OLTP) workloads. That is, sustaining simple lookup/update queries at high throughput. In contrast, modern hardware has since become orders of magnitude faster but also orders of magnitude more complex, and critical database applications now include—besides OLTP—the online analysis of huge data volumes stored in data warehouses, driven by tools that explore hidden trends in the data, such as online analytical processing (OLAP) tools that visualize databases as multidimensional cubes, and data mining tools that automatically construct knowledge models over these huge data-sets. This changed situation has recently made the research community realize that database architecture as it used to be is up for a full rewrite,[21] and to make future systems *self-tuning* to data distributions and workloads as they appear.[4]

In this paper, we summarize the work in the MonetDB[a] project that has focused on redefining database architecture by optimizing its major components (data storage, query processing algebra, query execution model, query processing algorithms, and query optimizer) toward better use of modern hardware in database applications that analyze large data volumes. One of the primary goals in this work has been breaking the memory wall.

Our focus here is the following innovations:

Vertical storage: Whereas traditionally, relational database systems store data in a row-wise fashion (which favors single record lookups), MonetDB uses columnar storage which favors analysis queries by better using CPU cache lines.

Bulk query algebra: Much like the CISC versus RISC idea applied to CPU design, the MonetDB algebra is deliberately simplified with respect to the traditional relational set algebra to allow for much faster implementation on modern hardware.

Cache-conscious algorithms: The crucial aspect in overcoming the memory wall is good use of CPU caches, for which careful tuning of memory access patterns is needed. This called for a new breed of query processing algorithms, of which we illustrate radix-partitioned hash-join in some detail.

Memory access cost modeling: For query optimization to work in a cache-conscious environment, we developed a methodology for creating cost models that takes the cost of memory access into account. In order to work on diverse computer architectures, these models are parameterized at runtime using automatic *calibration* techniques.

2. PRELIMINARIES

Computer architecture evolution in the past decades has had many facets. A general trend is that "latency lags bandwidth,"[16] which holds for both magnetic disk and RAM. This

[a] MonetDB is distributed using a nonrestrictive open-source license, see http://monetdb.cwi.nl

A previous version of this paper entitled "Database Architecture Optimized for the New Bottleneck: Memory Access" appeared in the *Proceedings of the International Conference on Very Large Data Bases* (September 1999), pp. 54–65.

has profoundly influenced the database area and indeed our work on MonetDB.

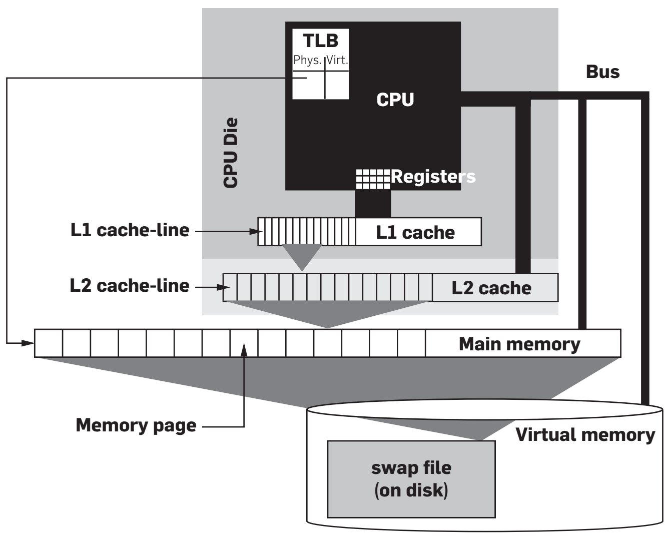
Another facet is that predictable array-wise processing models have been strongly favored in a string of recent CPU architectural innovations. While the rule "make the common case fast" was exploited time and time again to design and construct ever more complex CPUs, the difference in performance efficiency achieved by optimized code and intended use (e.g., "multimedia applications") versus nonoptimized code and nonintended use (e.g., "legacy database applications") has become very significant. A concrete example is the evolution of CPUs from executing a single instruction per clock cycle, to multi-issue CPUs that use deeply pipelined execution; sometimes splitting instructions in more than 30 dependent stages. Program code that has a high degree of independence and predictability (multimedia or matrix calculations) fills the pipelines of modern CPUs perfectly, while code with many dependencies (e.g., traversing a hash-table or B-tree) with unpredictable if-then-else checks, leaves many holes in the CPU pipelines, achieving much lower throughput.

2.1. The memory hierarchy

The main memory of computers consists of *dynamic random access memory* (DRAM) chips. While CPU clock-speeds have been increasing rapidly, DRAM access latency has hardly improved in the past 20 years. Reading DRAM memory took 1–2 cycles in the early 1980s, currently it can take more than 300 cycles. Since typically one in three program instructions is a memory load/store, this "memory wall" can in the worst case reduce efficiency of modern CPUs by two orders of magnitude. Typical system monitoring tools (top, or Windows Task manager) do not provide insight in this performance aspect, a 100% busy CPU could be 95% memory stalled.

To hide the high DRAM latency, the memory hierarchy has been extended with *cache memories* (cf., Figure 1), typically located on the CPU chip itself. The fundamental principle of all cache architectures is *reference locality*, i.e., the assumption that at any time the CPU repeatedly accesses only a limited amount of data that fits in the cache. Only the first access is "slow," as the data has to be loaded from main memory, i.e., a *compulsory cache miss*. Subsequent accesses (to the same data or memory addresses) are then "fast" as the data is then available in the cache. This is called a *cache hit*. The fraction of memory accesses that can be fulfilled from the cache is called *cache hit rate*.

Figure 1: Hierarchical memory architecture.



Cache memories are organized in *multiple cascading levels* between the main memory and the CPU. They become faster, but smaller, the closer they are to the CPU. In the remainder we assume a typical system with two cache levels (L1 and L2). However, the discussion can easily be generalized to an arbitrary number of cascading cache levels in a straightforward way.

In practice, cache memories keep not only the most recently accessed data, but also the instructions that are currently being executed. Therefore, almost all systems nowadays implement two separate L1 caches, a read-only one for instructions and a read-write one for data. The L2 cache, however, is usually a single "unified" read-write cache used for both instructions and data.

A number of fundamental characteristics and parameters of cache memories are relevant for the sequel:

Capacity (*C*). A cache's capacity defines its total size in bytes. Typical cache sizes range from 32KB to 4MB.

Line size (*Z*). Caches are organized in *cache lines*, which represent the smallest unit of transfer between adjacent cache levels. Whenever a cache miss occurs, a complete cache line (i.e., multiple consecutive words) is loaded from the next cache level or from main memory, transferring all bits in the cache line in parallel over a wide bus. This exploits spatial locality, increasing the chances of cache hits for future references to data that is "close to" the reference that caused a cache miss. The typical cache-line size is 64 bytes.

Associativity (*A*). An *A-way set associative* cache allows loading a line into one of *A* different positions. If $A > 1$, some *cache replacement* policy chooses one from the *A* candidates. *Least recently used* (*LRU*) is the most common replacement algorithm. In case $A = 1$, the cache is called *directly mapped*. This organization causes the least (virtually no) overhead in determining the cache-line candidate. However, it also offers the least flexibility and may cause a lot of so-called *conflict misses*. The other extreme case is *fully associative* caches. Here, each memory address can be loaded to any line in the cache ($A = \#$). This avoids conflict misses, and only so-called *capacity misses* occur as the cache capacity gets exceeded. However, determining the cache-line candidate in this strategy causes a relatively high overhead that increases with the cache size. Hence, it is feasible only for smaller caches. Current PCs and workstations typically implement two- to eight-way set associative caches.

Latency (λ) is the time span from issuing a data access until the result is available in the CPU. Accessing data that is already available in the L1 cache causes *L1 access latency* (λ_{L1}), which is typically rather small (1 or 2 CPU cycles). In case the requested data is not found in L1, an *L1 miss* occurs, additionally delaying the data access by *L2 access latency* (λ_{L2})

for accessing the L2 cache. Analogously, if the data is not yet available in L2, an *L2 miss* occurs, further delaying the access by *memory access latency* (λ_{Mem}) to finally load the data from main memory. Hence, the total latency to access data that is in neither cache is $\lambda_{\text{Mem}} + \lambda_{\text{L2}} + \lambda_{\text{L1}}$. As mentioned above, all current hardware actually transfers multiple consecutive words, i.e., a complete cache line, during this time.

Bandwidth (β) is a metric for the data volume (in megabytes) that can be transferred to the CPU per second. The different bandwidths are referred to as *L2 access bandwidth* (β_{L2}) and *memory access bandwidth* (β_{Mem}), respectively. Memory bandwidth used to be simply the cache-line size divided by the memory latency. Modern multiprocessor systems provide excess bandwidth capacity $\beta' \geq \beta$. To exploit this, caches need to be *nonblocking*, i.e., they need to allow more than one outstanding memory load at a time. CPUs that support out-of-order instruction execution can generate multiple concurrent loads, as the CPU does not block on a cache miss, but continues executing (independent) instructions. The number of outstanding memory requests is typically limited inside the CPU. The highest bandwidth in modern hardware is achieved if the access pattern is sequential; in which case the automatic memory prefetcher built into modern CPUs is activated. The difference between *sequential access bandwidth* ($\beta^s = \beta'$) and the respective (nonprefetched) *random access bandwidth* ($\beta^r = Z/\lambda^r$) can be a factor 10, which means that DRAM has truly become a block device, very similar to magnetic disk.

Transition lookaside buffer (**TLB**). A special kind of cache, the TLB is part of the virtual memory support built into modern CPUs: it remembers the latest translations of logical into physical page addresses (e.g., 64). Each memory load/store needs address translation; if the page address is in the TLB (a *TLB hit*), there is no additional cost. If not, a more complex lookup in a mapping table is needed; thus a *TLB miss* implies a penalty. Moreover, the lookup in the (memory-resident) TLB mapping table might generate additional CPU cache misses. Note that with a typical page size of 4KB and 64 entries in the TLB, on many systems TLB delay already comes into play for random access to data structures (e.g., hash-tables) larger than 256KB.

Unified hardware model. Summarizing the above discussion, we describe a computer's memory hardware as a cascading hierarchy of N levels of caches (including TLBs). An index $i \in \{1, \ldots, N\}$ identifies the respective value of a specific level. Exploiting the dualism that an access to level $i+1$ is caused by a miss on level i allows some simplification of the notation. Introducing the *miss latency* $l_i = \lambda_{i+1}$ and the respective *miss bandwidth* $b_i = \beta_{i+1}$ yields $l_i = Z_i/b_i$. Each cache level is characterized by the parameters given in Table 1.[b] We point out, that these parameters also cover the cost-relevant characteristics of disk accesses. Hence, viewing main memory (e.g., a database system's buffer pool) as cache (level $N+1$) for I/O operations, it is straightforward to include disk access in this hardware model.

[b] Costs for L1 accesses are assumed to be included in the CPU costs, i.e., λ_1 and β_1 are not used and hence undefined.

Table 1: Cache parameters per level ($i \in \{1, \ldots, N\}$)

Description	Unit	Symbol
Cache name (level)	–	L_i
Cache capacity	Bytes	C_i
Cache-line size	Bytes	Z_i
Number of cache lines	–	$\#_i = C_i/Z_i$
Cache associativity	–	A_i
Sequential ($x = s$) and random ($x = r$) access		
Miss latency	ns	l_i^x
Access latency	ns	$\lambda_{i+1}^x = l_i^x$
Miss bandwidth	Bytes/ns	$b_i^x = Z_i / l_i^x$
Access bandwidth	Bytes/ns	$\beta_{i+1}^x = b_i^x$

We developed a system-independent C program called *Calibrator*[c] to automatically measure these parameters on any computer hardware. The Calibrator uses carefully designed memory access patterns to generate cache misses in a controlled way. Comparing execution times of runs with and without cache misses, it can derive the cache parameters and latencies listed in Table 1. A detailed description of the Calibrator is given in Manegold.[11,12] Sample results for a PC with a 2.4 GHz Intel Core 2 Quad Q6600 CPU look as follows:

```
CPU loop + L1 access:  1.25 ns =  3 cy
Caches:
Level Size Linesize   Asso. Seq-miss-latency  rand-miss-latency
  1  32 KB   64 byte  4-way   0.91ns =  2 cy   4.74ns =  11 cy
  2   4 MB  128 byte  4-way  31.07ns = 75 cy  76.74ns = 184 cy
TLBs:
Level   #entries    pagesize     miss-latency
  1       256         4KB        9.00ns = 22 cy
```

3. MONETDB ARCHITECTURE

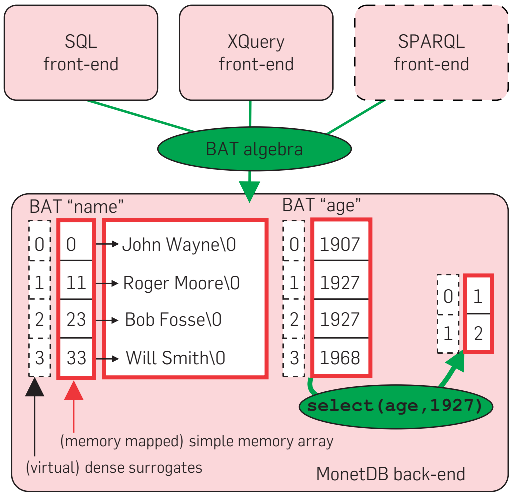
The storage model deployed in MonetDB is a significant deviation of traditional database systems. It uses the decomposed storage model (DSM),[8] which represents relational tables using vertical fragmentation, by storing each column in a separate `<surrogate,value>` table, called binary association table (BAT). The left column, often the surrogate or object-identifier (oid), is called the *head*, and the right column *tail*. MonetDB executes a low-level relational algebra called the *BAT algebra*. Data in execution is always stored in (intermediate) BATs, and even the result of a query is a collection of BATs.

Figure 2 shows the design of MonetDB as a back-end that acts as a BAT algebra virtual machine, with on top a variety of front-end modules that support popular data models and query languages (SQL for relational data, XQuery for XML).

BAT storage takes the form of two simple memory arrays, one for the head and one for the tail column (variable-width types are split into two arrays, one with offsets, and the other with all concatenated data). Internally, MonetDB stores

[c] http://www.cwi.nl/~manegold/Calibrator/calibrator.shtml

Figure 2: MonetDB: a BAT algebra machine.



columns using memory-mapped files. It is optimized for the typical situation that the surrogate column is a densely ascending numerical identifier (0, 1, 2,...); in which case the head array is omitted, and surrogate lookup becomes a fast array index read in the tail. In effect, this use of arrays in virtual memory exploits the fast in-hardware address to disk-block mapping implemented by the memory management unit (MMU) in a CPU to provide an $O(1)$ positional database lookup mechanism. From a CPU overhead point of view this compares favorably to B-tree lookup into slotted pages—the approach traditionally used in database systems for "fast" record lookup.

The Join and Select operators of the relational algebra take an arbitrary Boolean expression to determine the tuples to be joined and selected. The fact that this Boolean expression is specified at query time only, means that the RDBMS must include some *expression interpreter* in the critical runtime code-path of these operators. Traditional database systems implement each relational algebra operator as an iterator class with a *next*() method that returns the next tuple; database queries are translated into a pipeline of such iterators that call each other. As a recursive series of method calls is performed to produce *a single* tuple, computational interpretation overhead is significant. Moreover, the fact that the *next*() method of all iterators in the query plan is executed for each tuple, causes a large *instruction cache footprint*, which can lead to strong performance degradation due to instruction cache misses.[1]

In contrast, each BAT algebra operator has *zero degrees of freedom*: it does not take complex expressions as parameter. Rather, complex expressions are broken into a sequence of BAT algebra operators that perform one simple operation on an entire column of values ("bulk processing"). This allows the implementation of the BAT algebra to forsake an expression interpreting engine; rather all BAT algebra operations in the implementation map onto simple array operations. For instance, the BAT algebra expression

```
R:bat[:oid, :oid]:=select(B:bat[:oid,:int], V:int)
```

can be implemented at the C code level like:

```
for (i = j = 0; i <n; i++)
    if (B.tail[i] == V) R.tail[j++] = i;
```

The BAT algebra operators have the advantage that tight for-loops create high instruction locality which eliminates the instruction cache miss problem. Such simple loops are amenable to compiler optimization (loop pipelining, blocking, strength reduction), and CPU out-of-order speculation.

A potential danger of bulk processing is that it *materializes* intermediate results which in some cases may lead to excessive RAM consumption. Although RAM sizes increase quickly as well, there remain cases that we hit their limit as well. In the MonetDB/X100 project[3] it was shown how partial column-wise execution can be integrated into (nonmaterializing) pipelined query processing.

We can conclude that the MonetDB architecture for realizing database system functionality is radically different from many contemporary product designs, and the reasons for its design are motivated by opportunities for better exploiting modern hardware features.

4. CACHE-CONSCIOUS JOINS

Among the relational algebra operators, the *Join* operator, which finds all matching pairs between all tuples from two relations according to some Boolean predicate, is the most expensive operator—its complexity in the general case is quadratic in input size. However, for equality join predicates, fast (often linear) algorithms are available, such as *Hash-Join*, where the outer relation is scanned sequentially and a hash-table is used to probe the inner relation.

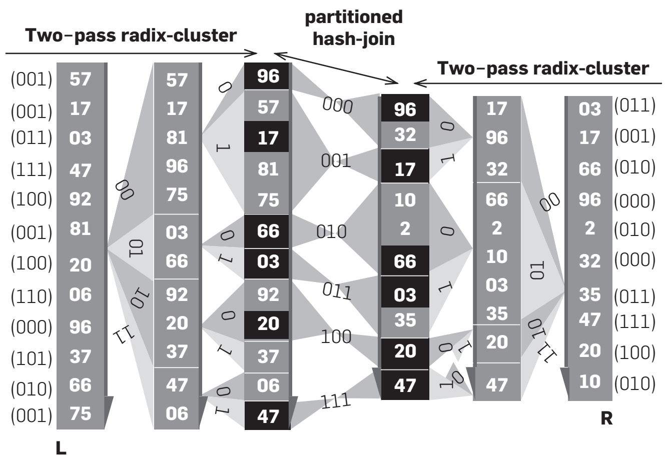
4.1. Partitioned hash-join

The very nature of the hashing algorithm implies that the access pattern to the inner relation (plus hash-table) is random. In case the randomly accessed data is too large for the CPU caches, each tuple access will cause cache misses and performance degrades.

Shatdal et al.[19] showed that a main-memory variant of Grace Hash-Join, in which both relations are first partitioned on hash-number into H separate *clusters*, that each fit into the L2 memory cache, performs better than normal bucket-chained hash-join. However, the clustering operation itself can become a cache problem: their straightforward clustering algorithm that simply scans the relation to be clustered once and inserts each tuple in one of the clusters, creates a random access pattern that writes into H separate locations. If H is too large, there are two factors that degrade performance. First, if H exceeds the number of TLB entries[d] each memory reference will become a *TLB miss*. Second, if H exceeds the number of available cache lines (L1

[d] If 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.

Figure 3: Partitioned hash-join ($H = 8 \Leftrightarrow B = 3$).



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

or L2), *cache thrashing* occurs, causing the number of cache misses to explode.

4.2. Radix-cluster

Our *Radix-Cluster* algorithm[2] divides a relation U into H clusters using multiple passes (see Figure 3). Radix-clustering on the lower B 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 bits ($\sum_1^P B_p = B$). The number of clusters created by the Radix-Cluster is $H = \Pi_1^P H_p$, where each pass subdivides each cluster into $H_p = 2^{B_p}$ new ones. When the algorithm starts, the entire relation is considered one single cluster, and is subdivided into $H_1 = 2^{B_1}$ clusters. The next pass takes these clusters and subdivides each into $H_2 = 2^{B_2}$ new ones, yielding $H_1 * H_2$ clusters in total, etc. With $P = 1$, Radix-Cluster behaves like the straightforward algorithm.

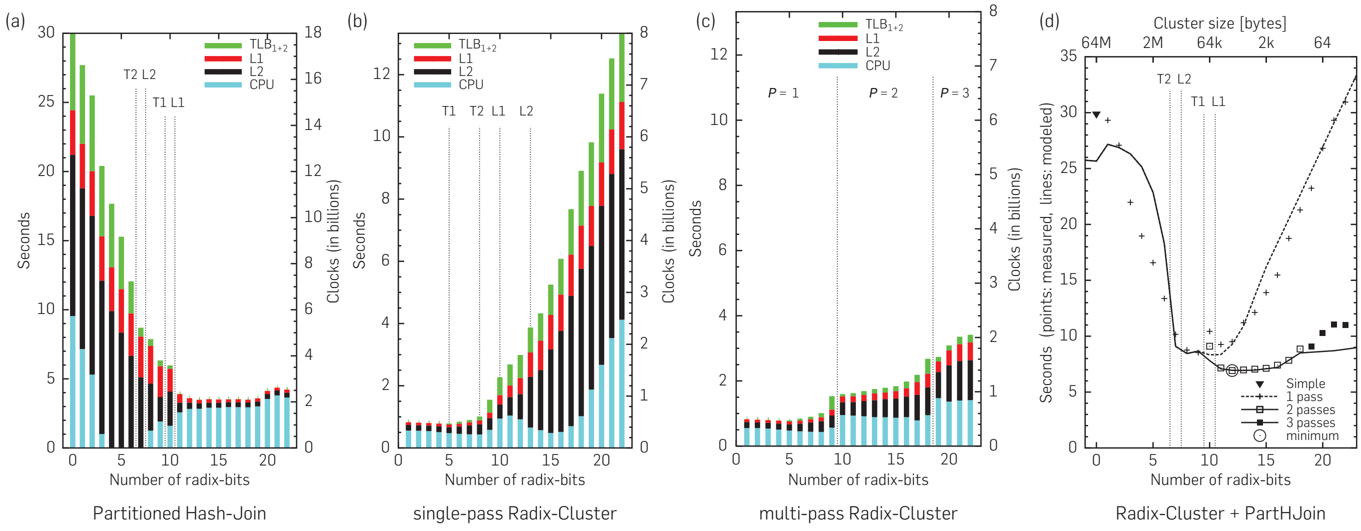
The crucial property of the Radix-Cluster is that the number of randomly accessed regions H_x can be kept low; while still a high overall number of H clusters can be achieved using multiple passes. More specifically, if we keep $H_x = 2^{B_x}$ smaller than the number of cache lines and the number of TLB entries, we completely avoid both TLB and cache thrashing. 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 clusters of $|U|/2^B$ tuples (with $|U|$ denoting the cardinality of the entire relation).

Figure 3 sketches a Partitioned Hash-Join of two integer-based relations L and R that uses two-pass Radix-Cluster to create eight clusters—the corresponding clusters are subsequently joined with Hash-Join. The first pass uses the two leftmost of the lower three bits to create four partitions. In the second pass, each of these partitions is subdivided into two partitions using the remaining bit.

For ease of presentation, we did not apply a hash-function in Figure 3. In practice, though, a hash-function should even be used on integer values to ensure that all bits of the join attribute play a role in the lower B bits used for clustering. Note that our surrogate numbers (oids) that stem from a dense integer domain starting at 0 have the property that the lowermost bits are the only relevant bits. Therefore, hashing is not required for such columns, and additionally, a Radix-Cluster on all log(N) relevant bits (where N is the maximum oid from the used domain) equals the well-known *radix-sort* algorithm.

Experiments. Figure 4 show experimental results for a Radix-Cluster powered Partitioned Hash-Join between two memory resident tables of 8 million tuples on an Athlon PC (see Manegold[13]). We used CPU counters to get a breakdown of cost between pure CPU work, TLB, L1, and L2 misses. The vertical axis shows time, while the horizontal axis varies the number of radix-bits B used for clustering (thus it is logarithmic scale with respect to the number of clusters H). Figure 4(a) shows that if a normal Hash-Join is used ($B = 0$), running time is more

Figure 4: Execution time breakdown of individual join phases and overall join performance.



than 30 s due to excessive L1, L2, and TLB misses, but if we join $2^{11} = 2048$ clusters of around 4000 tuples each (i.e., each cluster fits into the Athlon's L1 cache), performance improves around 10-fold. The lines T2, L2, T1, and L1 indicate the clustering degree after which the inner relation (plus hash-table) fits, respectively, the level 2 TLB, L2 data cache, level 1 TLB, and L1 data caches on this Athlon processor. However, Figure 4(b) shows that the straightforward clustering algorithm degrades significantly due to L1 and TLB misses after $B = 8$, as it is filling 256 clusters with only 256 L1 cache lines (on this Athlon), while for similar reasons L2 cache misses become a serious problem after 12 bits. To keep clustering efficient, we should therefore use multipass Radix-Cluster, as shown in Figure 4(c). Since using more clusters improves Partitioned Hash-Join yet degrades Radix-Cluster, the overall results in Figure 4(d) shows a sweet spot at $B = 12$ (two passes).

When a user submits a query to a running database server, its query optimizer determines a physical plan, choosing the right order of the operators as well as choosing the physical algorithm to use. For instance, it may compare SortMerge-with Hash-Join. Additionally, in case of Hash-Join, the optimizer must now also determine how many partitions H, thus, radix-bits B, to use. On the one hand, it needs crucial parameters of the unified hardware model (i.e., the cache configurations) as derived by Calibrator (see Section 2.1); e.g., at DBMS startup. On the other hand, it should model the memory access cost of query processing operators given a value distribution estimate and tuning parameters (such as B). The lines in Figure 4(d) represent the cost prediction of our model for Partitioned Hash-Join, indicating that the techniques described in Section 5 can be quite accurate.

5. MODELING MEMORY ACCESS COSTS

Cache-conscious database algorithms, such as the radix-partitioned hash-join, achieve their optimal performance only if they are carefully tuned to the hardware specifics. Predictive and accurate cost models provide the cornerstones to automate this tuning task. We model the data access behavior in terms of a combination of basic access patterns using the unified hardware model from Section 2.1.

5.1. Memory access cost

Memory access cost can be modeled by estimating the number of cache misses M and scoring them with their respective miss latency l.[13] Akin to detailed I/O cost models we distinguish between random and sequential access. However, we now have multiple cache levels with varying characteristics. Hence, the challenge is to predict the number and kind of cache misses *for all cache levels*. Our approach is to treat all cache levels individually, though equally, and calculate the total cost as the sum of the cost for all levels:

$$T_{\text{Mem}} = \sum_{i=1}^{N} (\mathrm{M}_i^{\mathrm{s}} \cdot l_i^{\mathrm{s}} + \mathrm{M}_i^{\mathrm{r}} \cdot l_i^{\mathrm{r}}).$$

This leaves the challenge to properly estimate the number and kind of cache misses per cache level for various database algorithms. The task is similar to estimating the number and kind of I/O operations in traditional cost models. However, our goal is to provide a generic technique for predicting cache miss rates, sacrificing as little accuracy as possible.

The idea is to abstract data structures as *data regions* and model the complex data access patterns of database algorithms in terms of simple compounds of a few *basic data access patterns*. For these basic patterns, we then provide cost functions to estimate their cache misses. Finally, we present rules to combine basic cost functions and to derive the cost functions of arbitrarily complex patterns.

5.1.1. Basic Access Patterns

Data structures are modeled using a set of *data regions* $\mathbb{D}$. A data region $R \in \mathbb{D}$ consists of $|R|$ *data items* of size $\underline{R}$ (in bytes). We call $|R|$ the *length* of region R, $\underline{R}$ its *width*, and $||R|| = |R| \cdot \underline{R}$ its *size*.

A database table is hence represented by a region R with $|R|$ being the table's cardinality and $\underline{R}$ being the tuple size (width). Similarly, more complex structures like trees are modeled by regions with $|R|$ representing the number of nodes and $\underline{R}$ representing the size (width) of a node.

The following basic access patterns are eminent in the majority of relational algebra implementations.

A **single sequential traversal** `s_trav`(R) sweeps over R, accessing each data item in R exactly once (cf., Figure 5).

A **single random traversal** `r_trav`(R) visits each data item in R exactly once. However, the data items are not accessed in storage order, but chosen randomly (cf., Figure 6).

A **repetitive sequential traversal** `rs_trav`(r, d, R) performs r sequential traversals over R. d = uni (*unidirectional*) indicates that all traversals sweep over R in the same direction. d = bi (*bidirectional*) indicates that subsequent traversals go in alternating directions.

A **repetitive random traversal** `rr_trav`(r, R) performs r random traversals over R. Assuming that the permutation orders of two subsequent traversals are independent, there is no point in discriminating uni- and bidirectional accesses.

Random access `r_acc`(r, R) hits r randomly chosen data items in R after another. The choices are independent of each other. Each data item may be hit more than once. Even

Figure 5: Single sequential traversal: `s_trav(R)`.

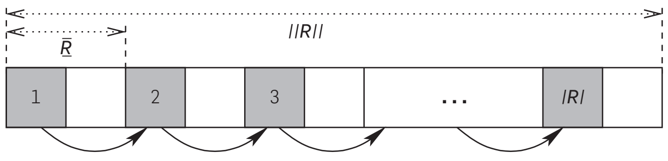
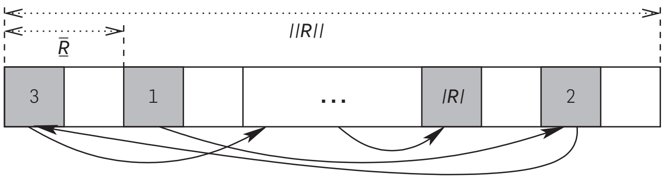


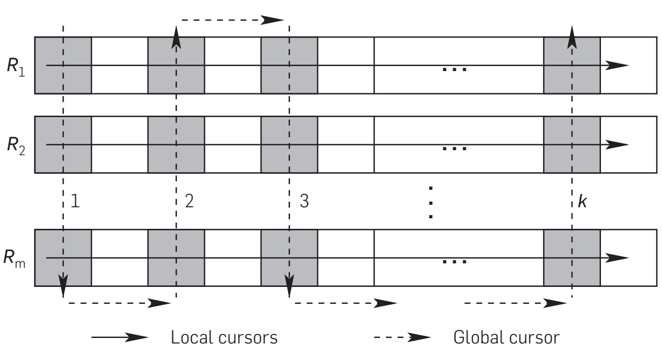
Figure 6: Single random traversal: `r_trav(R)`.



with $r \geq |R|$ we do not require that each data item is hit at least once.

An **interleaved access** nest($R, m, \mathcal{P}, O[, D]$) models a nested multicursor access pattern where R is divided into m (equal-sized) subregions. Each subregion has its own local cursor. All local cursors perform the same basic pattern $\mathcal{P}$. O specifies, whether the global cursor picks the local cursors randomly (O = ran) or sequentially (O = seq). In the latter case, D specifies, whether all traversals of the global cursor across the local cursors use the same direction (D = uni), or whether subsequent traversals use alternating directions (D = bi). Figure 7 shows an example.

Figure 7: Interleaved multicursor access: nest(*R*, *m*, `s_trav(R)`, `seq`, `bi`).



5.1.2. Compound Access Patterns

Database operations access more than one data region, e.g., their input(s) and their output, which leads to *compound* data access patterns. We use $\mathbb{P}_b$, $\mathbb{P}_c$, and $\mathbb{P} = \mathbb{P}_b \cup \mathbb{P}_c$ ($\mathbb{P}_b \cap \mathbb{P}_c = \emptyset$) to denote the set of basic access patterns, compound access patterns, and all access patterns, respectively.

Be $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathbb{P}$ ($p > 1$) data access patterns. There are only two principle ways to combine patterns. They are executed either *sequentially* ($\oplus : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$) or *concurrently* ($\odot : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$). We can apply $\oplus$ and $\odot$ repeatedly to describe more complex patterns.

Table 2 illustrates compound access patterns of some typical database algorithms. For convenience, reoccurring compound access patterns are assigned a new name.

5.2. Cost functions

For each basic pattern, we estimate both sequential and random cache misses for each cache level $i \in \{1, \ldots, N\}$. Given an access pattern $\mathcal{P} \in \mathbb{P}$, we describe the number of misses per cache level as a pair

$$\vec{\mathrm{M}}_i(\mathcal{P}) = \left\langle \mathrm{M}_i^{\mathrm{s}}(\mathcal{P}), \mathrm{M}_i^{\mathrm{r}}(\mathcal{P}) \right\rangle \in \mathsf{N} \times \mathsf{N}$$

containing the number of sequential and random cache misses. The detailed cost functions for all basic patterns introduced above can be found in Manegold.[11,12]

The major challenge with compound patterns is to model cache interference and dependencies among basic patterns.

5.2.1. Sequential Execution

When executed sequentially, patterns do not interfere. Consequently, the resulting total number of cache misses is at most the sum of the cache misses of all patterns. However, if two subsequent patterns operate on the same data region, the second might benefit from the data that the first one leaves in the cache. It depends on the cache size, the data sizes, and the characteristics of the patterns, how many cache misses may be saved this way. To model this effect, we consider the contents or *state* of the caches, described by a set $\mathbf{S}$ of pairs $\langle R, \rho \rangle \in \mathbb{D} \times [0, 1]$, stating for each data region R the fraction ρ that is available in the cache.

In Manegold[11,12] we discuss how to calculate (i) the cache misses of a basic pattern $\mathcal{P}_q \in \mathbb{P}_b$ given a cache state $\mathbf{S}^{q-1}$ as

$$\vec{\mathrm{M}}_i\,(\mathrm{S}_i^{q-1}, \mathcal{P}_q) = \mathcal{F}'\,(\mathrm{S}_i^{q-1}, \vec{\mathrm{M}}_i(\mathcal{P}_q)),$$

and (ii) the resulting cache state after executing $\mathcal{P}_q$ as

$$\mathrm{S}_i^q\,(\mathrm{S}_i^{q-1}, \mathcal{P}_q) = \mathcal{F}''(\mathrm{S}_i^{q-1}, \mathcal{P}_q).$$

Table 2: Sample data access patterns (*U*, *V*, *V'*, *W* $\in \mathbb{D}$)

Algorithm	Pattern Description	Name
$W \leftarrow \mathit{select}(U)$	$\texttt{s\_trav}(U) \odot \texttt{s\_trav}(W)$	
$W \leftarrow \mathit{nested\_loop\_join}(U, V)$	$\texttt{s\_trav}(U) \odot \texttt{rs\_trav}(\lvert U\rvert, \texttt{uni}, V) \odot \texttt{s\_trav}(W)$	$=: \texttt{nl\_join}(U, V, W)$
$W \leftarrow \mathit{zig\_zag\_join}(U, V)$	$\texttt{s\_trav}(U) \odot \texttt{rs\_trav}(\lvert U\rvert, \texttt{bi}, V) \odot \texttt{s\_trav}(W)$	
$V' \leftarrow \mathit{hash\_build}(V)$	$\texttt{s\_trav}(V) \odot \texttt{r\_trav}(V')$	$=: \texttt{build\_hash}(V, V')$
$W \leftarrow \mathit{hash\_probe}(U, V')$	$\texttt{s\_trav}(U) \odot \texttt{r\_acc}(\lvert U\rvert, V') \odot \texttt{s\_trav}(W)$	$=: \texttt{probe\_hash}(U, V', W)$
$W \leftarrow \mathit{hash\_join}(U, V)$	$\texttt{build\_hash}(V, V') \oplus \texttt{probe\_hash}(U, V', W)$	$=: \texttt{h\_joins}(U, V, W)$
$\{U_j\}\rvert_{j=1}^{m} \leftarrow \mathit{cluster}(U, \mathrm{m})$	$\texttt{s\_trav}(U) \odot \texttt{nest}(\{U_j\}\rvert_{j=1}^{m}, m, \texttt{s\_trav}(U_j), \texttt{ran})$	$=: \texttt{part}(U, m, \{U_j\}\rvert_{j=1}^{m})$
$W \leftarrow \mathit{part\_nl\_join}(U, V, \mathrm{m})$	$\texttt{part}(U, m, \{U_j\}\rvert_{j=1}^{m}) \oplus \texttt{part}(V, m, \{V_j\}\rvert_{j=1}^{m}) \oplus \texttt{nl\_join}(U_1, V_1, W_1) \oplus \ldots \oplus \texttt{nl\_join}(U_m, V_m, W_m)$	
$W \leftarrow \mathit{part\_h\_join}(U, V, \mathrm{m})$	$\texttt{part}(U, m, \{U_j\}\rvert_{j=1}^{m}) \oplus \texttt{part}(V, m, \{V_j\}\rvert_{j=1}^{m}) \oplus \texttt{nl\_join}(U_1, V_1, W_1) \oplus \ldots \oplus \texttt{h\_join}(U_m, V_m, W_m)$	

With these, we can calculate the number of cache misses that occur when executing patterns $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathbb{P}$, $p > 1$ sequentially, given an initial cache state $\mathbf{S}^0$, as

$$\vec{M}_i\,(S_i^0, \oplus(\mathcal{P}_1,\ldots,\mathcal{P}_p)) = \sum_{q=1}^{p} \vec{M}_i\,(S_i^{q-1}, \mathcal{P}_q).$$

5.2.2. Concurrent Execution

When executing patterns concurrently, we actually have to consider the fact that they are competing for the same cache. We model the impact of the cache interference between concurrent patterns by dividing the cache among all patterns. Each pattern $\mathcal{P}$ gets a fraction $0 < .\nu < 1$ of the cache according to its *footprint size* $\mathbf{F}$, i.e., the number of cache lines that it potentially revisits. The detailed formulas for $\mathbf{F}_i(\mathcal{P})$ with $\mathcal{P} \in \mathbb{P}$ are given in Manegold.[11,12]

We use $\vec{\mathbf{M}}_{i/\nu}$ to denote the number of misses with only a fraction $0 < .\nu < 1$ of the total cache size available.

With these tools at hand, we calculate the cache misses for concurrent execution of patterns $\mathcal{P}_1, \ldots, \mathcal{P}_p \in \mathbb{P}$ $(p > 1)$ given an initial cache state $\mathbf{S}^0$ as

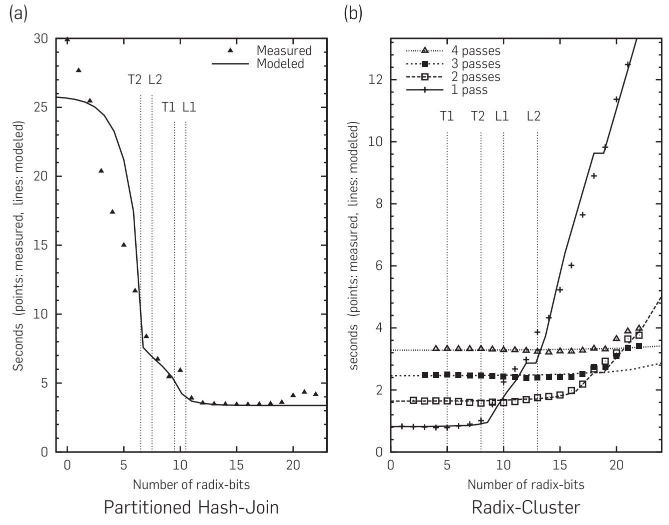
$$\vec{M}_i\,(S_i^0, \odot\,(\mathcal{P}_1, \ldots, \mathcal{P}_p)) = \sum_{q=1}^{p} \vec{M}_{i/\nu_q}\,(S_i^0, \mathcal{P}_q).$$

For our radix-partitioned hash-join algorithm, Figures 4d and 8 compare the cost predicted by our cost model to the measured execution times on an Athlon PC. An exhaustive experimental validation of our models is presented in Manegold.[11,12]

5.2.3. Query Execution Plans

With the techniques discussed, we have the basic tools at hand to estimate the number and kind of cache misses of complete query plans, and hence can predict their memory access costs. The various operators in a query plan are combined in the same way the basic patterns are combined to form compound patterns. Basically, the query plan describes, which operators are executed one after the other and which are executed concurrently. We view pipelining as concurrent execution of data-dependent operators. Hence, we can derive the complex memory access pattern of a query plan by combining the compound patterns of the operators as discussed above. Considering the caches' states as introduced before takes care of handling data dependencies.

Figure 8: Sample cost model validation.



6. RELATED WORK

The growing mismatch between the way database systems are engineered versus hardware evolution was first brought to light in a number of workload studies. An early study[15] already showed database workloads, compared with scientific computation, to exhibit significantly more instruction cache misses (due to a large code footprint) and more (L2) data cache misses.

Instruction cache misses are specifically prevalent in transaction processing workloads. The STEPS[10] approach therefore organizes multiple concurrent queries into execution teams, and evaluates each query processing operator for all members of the team one after another, while its code is hot. Another proposal in this direction, aimed at analysis queries, proposed to split query plans into sections whose code fits the instruction cache, putting a so-called "Buffer" operator on the section boundary.[23] The Buffer operator repeatedly invoke the query subsection below it, buffering the resulting tuples without passing them on yet, such that the operators in the subsection are executed multiple times when hot, amortizing instruction cache misses. The high locality of the BAT algebra operators in MonetDB and materialization of results can be seen as an extreme form of this latter strategy.

In the area of index structures, Cache-sensitive B+ Trees (CSB+-Trees)[17] ensure that internal nodes match the cache-line size, optimizing the number of cache-line references, and introduce highly optimized in-node search routines for faster lookup.

The MonetDB work[2,12,13] showed vertical data fragmentation (DSM[8]) to benefit analysis queries, due to reduced memory traffic and an increased spatial locality. Column-stores have since received much attention for use in data warehousing environments (e.g., C-Store,[20] and the CWI follow-up system MonetDB/X100[3]), introducing column-store specific compression and query evaluation techniques.

Considering hash-join, cache-sized partitioning was first proposed in Shatdal[19] and subsequently improved in Boncz,[2] as summarized in Section 4. The Radix-Cluster algorithm was later supplemented with an inverse Radix-Decluster algorithm,[14] that allows to perform arbitrary data permutations in a cache-efficient manner (this can be used for sorting, as well as for postponing the propagation of join columns to after the join phase).

An alternative hash-join approach uses software prefetching, exploiting the explicit memory-to-cache prefetching instructions offered by modern CPUs. Group prefetching was shown in Chen[6] to perform better than cache-partitioning and was also shown to be more resistant to interference by other programs. Prefetching was also successfully applied in B-tree access[7] to increase the width of the nodes without paying the latency cost of fetching the additional cache lines. Memory prefetching has also been applied to optimize various data accesses in the Inspector Join algorithm.[5] A general disadvantage of hardware prefetching is that it is notoriously platform-dependent and difficult to tune, therefore hindering its application in generic software packages. A precondition for such tuning is the availability of a unified hardware model that provides parameters, and memory cost formulas, as introduced in Manegold[11,12] and summarized in Section 5.

Architecture-conscious results continue to appear regularly in major database research publications, and also have a specialized workshop (DaMoN) colocated with SIGMOD. Other topics that have been addressed include minimizing branch mispredictions in selection operations,[18] using SIMD instructions for database tasks,[22] and query processing with GPU hardware,[9] which led in 2006 to a NVIDIA graphics card to become the PennySort sorting benchmark champion. Recently there is interest in the use of Flash memory for database storage as well as query parallelization for multicore CPUs.

7. CONCLUSION

When MonetDB debuted more than a decade ago, the idea of using vertical storage was radical, however in the past few years the database community has confirmed its benefits over horizontal-only storage,[20] and the principle is currently being adopted broadly in commercial systems.

Less repeated, as of yet, have been the MonetDB results that focus on highly CPU-efficient execution. The reshaping of relational algebra to map it into tightly looped array processing, leads to as yet unmatched raw computational efficiency, benefiting from trends in CPU design and compiler optimizer support.

In the broad sense, the research around MonetDB aims at redefining database architecture in the face of an ever-changing computer architecture and database application landscape. This research still continues, for instance by making database systems *self-tuning* using automatic on-the-fly indexing strategies that piggyback on query execution ("database cracking"), and by improving query optimizer efficiency and robustness using a modular *runtime* framework that transforms query optimization from a static procedure that precedes query execution, into a dynamic mechanism where query optimization and execution continuously interleave.

References

1. Ailamaki, A.G., DeWitt, D.J., Hill, M.D., and Wood, D.A. DBMSs on a modern processor: Where does time go? In *International Conference on Very Large Data Bases (VLDB)*, Sept. 1999, 266–277.
2. Boncz, P.A., Manegold, S., and Kersten, M.L. Database architecture optimized for the new bottleneck: Memory access. In *International Conference on Very Large Data Bases (VLDB)*, Sept. 1999, 54–65.
3. Boncz, P.A., Zukowski, M., and Nes, N. MonetDB/X100: Hyper-pipelining query execution. In *International Conference on Innovative Data Systems Research (CIDR)*, Jan. 2005, 225–237.
4. Chaudhuri, S. and Weikum, G. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *International Conference on Very Large Data Bases (VLDB)*, Sept. 2000, 1–10.
5. Chen, S., Ailamaki, A., Gibbons, P.B., and Mowry, T.C. Inspector joins. In *International Conference on Very Large Data Bases (VLDB)*, Aug. 2005.
6. Chen, S., Ailamaki, A., Gibbons, P.B., and Mowry, T.C. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, *32*, 3 (2007).
7. Chen, S., Gibbons, P.B., and Mowry, T.C. Improving index performance through prefetching. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2001.
8. Copeland, G.P. and Khoshafian, S. A decomposition storage model. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, May 1985, 268–279.
9. Govindaraju, N., Gray, J., Kumar, R., and Manocha, D. GPUTeraSort: High performance graphics co-processor sorting for large database management. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2006.
10. Harizopoulos, S. and Ailamaki, A. STEPS towards cache-resident transaction processing. In *International Conference on Very Large Data Bases (VLDB)*, Aug. 2004.
11. Manegold, S. Understanding, modeling, and improving main-memory database performance. PhD thesis, Universiteit van Amsterdam, Amsterdam, the Netherlands, Dec. 2002.
12. Manegold, S., Boncz, P.A., and Kersten, M.L. Generic database cost models for hierarchical memory systems. In *International Conference on Very Large Data Bases (VLDB)*, Aug. 2002, 191–202.
13. Manegold, S., Boncz, P.A., and Kersten, M.L. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, *14*, 4 (July 2002), 709–730.
14. Manegold, S., Boncz, P.A., Nes, N., and Kersten, M.L. Cache-conscious radix-decluster projections. In *International Conference on Very Large Data Bases (VLDB)*, Aug. 2004, 684–695.
15. Maynard, A.M.G., Donnelly, C.M., and Olszewski, B.R. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. *SIGOPS Oper. Syst. Rev.*, *28*, 5 (Aug. 1994), 145–156.
16. Patterson, D. Latency lags bandwidth. *Commun. ACM 47*, 10 (Oct. 2004), 71–75.
17. Rao, J. and Ross, K.A. Making B+ -Trees cache conscious in main memory. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2000.
18. Ross, K.A. Conjunctive selection conditions in main memory. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, June 2002.
19. Shatdal, A., Kant, C., and Naughton, J. Cache conscious algorithms for relational query processing. In *International Conference on Very Large Data Bases (VLDB)*, Sept. 1994, 510–512.
20. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S.R., O'Neil, E.J., O'Neil, P.E., Rasin, A., Tran, N., and Zdonik, S.B. C-Store: A column-oriented DBMS. In *International Conference on Very Large Data Bases (VLDB)*, Sept. 2005, 553–564.
21. Stonebraker, M., Madden, S.R., Abadi, D.J., Harizopoulos, S., Hachem, N., and Helland, P. The end of an architectural era (it's time for a complete rewrite). In *International Conference on Very Large Data Bases (VLDB)*, Sept. 2007, 1150–1160.
22. Zhou, J. and Ross, K.A. Implementing database operations using SIMD instructions. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2002.
23. Zhou, J. and Ross, K.A. Buffering database operations for enhanced instruction cache performance. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, June 2004.

Peter A. Boncz (Peter.Boncz@cwi.nl) CWI, Kruislaan, Amsterdam, the Netherlands.

Martin L. Kersten (Martin.Kersten@cwi.nl) CWI, Kruislaan, Amsterdam, the Netherlands.

Stefan Manegold (Stefan.Manegold@cwi.nl) CWI, Kruislaan, Amsterdam, the Netherlands.