Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct

Stefan Manegold Martin L. Kersten Peter Boncz
CWI, Amsterdam, The Netherlands
{manegold,mk,boncz}@cwi.nl

ABSTRACT

The holy grail for database architecture research is to find a solution that is *Scalable & Speedy*, to run on anything from small ARM processors up to globally distributed compute clusters, *Stable & Secure*, to service a broad user community, *Small & Simple*, to be comprehensible to a small team of programmers, *Self-managing*, to let it run out-of-the-box without hassle.

In this paper, we provide a trip report on this quest, covering both past experiences, ongoing research on hardware-conscious algorithms, and novel ways towards self-management specifically focused on column store solutions.

1. INTRODUCTION

Forty years of relational database technology seemingly converged into one commonly accepted system construction lore. It was based on a tuple-at-a-time pipelined execution paradigm, ARIES transactions and query optimization based on (mostly) static cost models, and targeting minimization of hard disk accesses. As RDBMS functionality reached a plateau, system complexity had increased considerably, leading to attempts to reduce the tuning knobs of database systems using, e.g., design wizards.

Meanwhile, the hardware landscape changed considerably. High-performance application requirements shifted from transaction processing to requirements posed by, e.g., business intelligence applications. Similarly, data being managed changed from relational tables only through object-oriented to XML and RDF, putting increasing complexity onto existing systems. These trends combined, caused a major shift in how database management solutions can be crafted, and by now it has become clear that the traditional lore is just one local minimum in the overall design space.

The abundance of main memory makes it the prime choice for current database processing. However, effective use of CPU caches became crucial. Designing a DBMS from the perspective of large main memories and multiple data models with different query languages called for a re-examination of the basic storage structures needed. Column-stores have become a crucial piece in this puzzle, not only because they reduce the amount of data manipulated within a database engine, but also because columns form an ideal building block for realizing more complex structures (such as tables, but also objects and even trees or graphs).

In this paper, we summarize the quest to renovate database architecture with the goal of addressing data management needs with a common core rather than bolting on new subsystems [38], and pushing through the functionality plateau, especially using self-adaptation, exploiting resource optimization opportunities as they appear dynamically, rather than by static prediction. This quest covers the major design space choices, experiences gained and ongoing research around the MonetDB system.

In all, this story demonstrates the viability of new self-adaptive life-forms that may be very successful yet are very different from the long existing species. In this, the MonetDB strain forms an open platform available for all who want to explore further evolution in the data management ecosystem.

2. GENETICS OF COLUMN STORES

Column stores have a long history and come in many variations [35]. As far back as in the 1970's, column storage schemes for relational data have been used mostly for statistical applications.

The heart of a column-store is a direct mapping of a relational table into its canonical binary (2-ary) equivalent [11] holding an imaginary system surrogate OID and a value. While the binary relations can be seen as tables, it is evident that storing and optimizing them as relational tables in existing engines fails. For example, their per tuple storage overhead is substantial and the cost-based optimizers can not cope with the search space explosion. Instead, a column-store specific representation opens opportunities for selective replication, e.g., using different sort orders, and more effective compression schemes. Such structural changes percolate through the DBMS software stack, including the way the engine operates.

For decades, the database execution paradigm has been a tuple-at-a-time approach, geared at reducing the intermediate storage footprint of queries. With shifting to larger main memories, it becomes possible to explore a column-at-a-time approach. Column-wise execution is more efficient on current hardware, which incidentally looks much like the super-computers of the past (based on deep pipelining and SIMD vector execution). The abundance of large numbers of cores allow also for experimentation with the function-at-a-core approach [16, 4].

Likewise, improving access to pieces of interest is traditionally addressed by maintaining indices exploited by the inner-core relational operators. The next improvement came from materialized views, which gives the query optimizer alternatives to reduce access cost. Both, however, are maintained under (heavy) updates and require DBA expertise [3, 42]. An alternative pioneered in column stores is to aim for just-in-time (partial) indexing, based on the rationale that not all data is equally important.

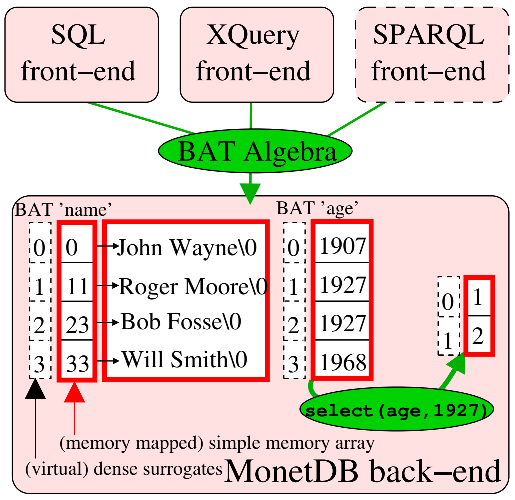

Figure 1: MonetDB: a BAT Algebra Machine

Together, the trends triggered an interest into the re-design of the basic relational algebra operators using novel hardware, as reported in a series of successful workshops [4].

The MonetDB system uses the column-at-a-time paradigm, just-in-time indexing, plan-for-reuse, where optimization opportunities are helped by its columnar result materialization. In the derivative X100 project [43], it was shown that the efficiency of columnar execution can be retained and even enhanced in a pipelined execution model without materialization, scaling to huge disk-based scenarios by exploiting vector processing and light-weight compression.

3. MONETDB ARCHITECTURE

The storage model deployed in MonetDB is a significant deviation of traditional database systems. It uses the Decomposed Storage Model (DSM) [11], which represents relational tables using vertical fragmentation, by storing each column in a separate <`surrogate,value`> table, called BAT (Binary Association Table). The left column, often the surrogate or `oid` (object-identifier), 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 1 shows the design of MonetDB as a back-end that acts as a BAT Algebra virtual machine programmed with the MonetDB Assembler Language (MAL). The top consists of a variety of query language compilers that produce MAL programs, e.g., 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 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 MMU (memory management unit) 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 [6].

In contrast, each BAT Algebra operator maps to a simple MAL instruction, which 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 each perform a 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.

3.1 Optimizer Architecture

MonetDB's query processing scheme is centered around three software layers. The top is formed by the query language parser and a heuristic, language- and data model-specific optimizer to reduce the amount of data produced by intermediates and to exploit catalogue knowledge on join-indices. The output is a logical plan expressed in MAL.

The second tier consists of a collection of optimizer modules, which are assembled into optimization pipelines. The modules provide facilities ranging from symbolic processing up to just-in-time data distribution and execution. MAL programs are transformed into more efficients ones and sprinkled with resource management directives. The approach breaks with the hitherto omnipresent cost-based optimizers by recognition that not all decisions can be cast together in a single cost formula. Operating on the common binary-relational back-end algebra, these optimizer modules are shared by all front-end data models and query languages.

The third tier, the MAL interpreter, contains the library of highly optimized implementation of the binary relational algebra operators. They maintain properties over the object accessed to gear the selection of subsequent algorithms. For example, the Select operator can benefit both from sorted-ness of the BAT or it may call for a sample to derive the expected sizes. [1]

3.2 Front-ends

The original DSM paper [11] articulates the idea that DSM could be the physical data model building block to empower many more

[1]More details can be found on-line at `http://www.monetdb.com/`

complex user-level data models. This observation is validated with the open-source MonetDB architecture, where all front-ends produce code for the same columnar back-end. We briefly discuss how BATs are used for processing widely different front-end data models and their query languages.

SQL. The relational front-end decomposes tables by column, in BATs with a dense (non-stored) TID head, and a tail column with values. For each table, a BAT with deleted positions is kept. Delta BATs are designed to delay updates to the main columns, and allow a relatively cheap snapshot isolation mechanism (only the delta BATs are copied). MonetDB/SQL also keeps additional BATs for join indices; and value indices are created on-the-fly.

XQuery. The work in the Pathfinder project [8] makes it possible to store XML tree structures in relational tables as <pre,post> coordinates, represented as a collection of BATs. In fact, the prenumbers are densely ascending, hence can be represented as a (non-stored) dense TID column, saving storage space and allowing fast O(1) lookups. Only slight extensions to the BAT Algebra were needed, in particular a series of region-joins called staircase joins were added to the system for the purpose of accelerating XPath predicates. MonetDB/XQuery provides comprehensive support for the XQuery language, the XQuery Update Facility, and a host of IR-specific extensions.

Arrays. The Sparse Relational Array Mapping (SRAM) project maps large (scientific) array-based data-sets into MonetDB BATs, and offers a high-level comprehension-based query language [12]. This language is subsequently optimized on various levels before being translated into BAT Algebra. Array front-ends are particularly useful in scientific applications.

SPARQL. The MonetDB team has started development to provide efficient support for the W3C query language SPARQL, using MonetDB as a scalable RDF storage. Preliminary experimental performance results were presented in [36].

4. CACHE-CONSCIOUS ALGORITHMS

Database systems grind massive amounts of data to find the snippets or relationships of interest. This amounts to a sizable dataflow within most architectures, which is countered by good query plan optimizers and navigational aids from, e.g., indices and materialized views. However, in the end, information is passed through the memory hierarchy to the processor pool. Both puts a limit on what can be achieved.

The seminal paper by Ailamaki [6] showed that database systems exploit hardware poorly. A new approach based on the recognition of the internal hardware limitations was urgently needed. We illustrate it with a synopsis of cache-conscious join algorithms, the core of our VLDB 1999 paper [9] as well as some follow-up work.

4.1 Partitioned Hash-join

The nature of any 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. [34] 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 each

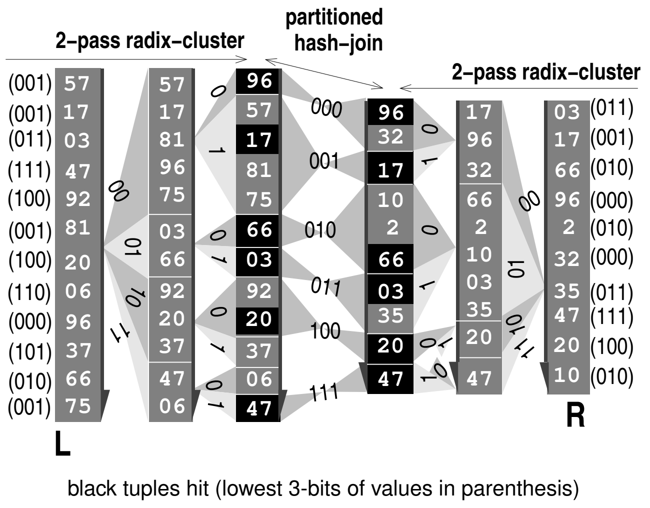

Figure 2: Partitioned Hash-Join (H=8 ⇔ B=3)

memory reference will become a *TLB miss*. Second, if H exceeds the number of available cache lines (L1 or L2), *cache thrashing* occurs, causing the number of cache misses to explode.

4.2 Radix-cluster

Our *Radix-Cluster* algorithm [9] divides a relation U into H clusters using multiple passes (see Figure 2). 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 $H = \prod_1^P H_p$, where each pass subdivides each cluster into $H_p = 2^{B_p}$ new ones. 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 2 sketches a Partitioned Hash-Join of two integer-based relations L and R that uses two-pass Radix-Cluster to create 8 clusters – the corresponding clusters are subsequently joined with Hash-Join. The first pass uses the 2 left-most of the lower 3 bits to create 4 partitions. In the second pass, each of these partitions is sub-divided into 2 partitions using the remaining bit. Experimental assessment confirms that trading extra CPU work for reduced random memory access with multi-pass Radix-Cluster signifiantly improves performance on modern hardware [9].

In [25], we conclude that cache-conscious algorithms achieve their full performance only once next to memory access also CPU costs are minimized, e.g., by removing function calls and divisions (in the hash function) from inner-most loops. Extensive experiments show that memory and CPU optimization boost each other, i.e., their combined improvement is larger than the sum of their individual improvements. CPU- and cache-optimized radix-clustered partitioned hash-join can easily achieve an order of magnitude performance improvement over simple hash-join.

4.3 Radix-decluster

Joins in real-life queries almost always come with projections over other attributes. In a column store this means that, at some point, we have to reconstruct the complete tuple. This leads to a 2-phase algorithm: join index construction and column projection. In the first phase, we access the columns with the join attributes, find matching pairs of tuples, and keep them as a join-index [39]. In the second phase, all projection columns are accessed one-by-one and a result column is produced using a join-index to fetch values from the input column.

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

Our Radix-Decluster algorithm presented in [28] is the crucial tool of MonetDB to process (i.e., join, but also re-order) huge tables with a good access pattern, both in terms of CPU cache access as well as I/O access (through virtual memory).

In our experiments, we tested various cache-conscious join (projection) strategies both on the traditional storage layout (NSM) and DSM storage schemes. One important conclusion from these experiments is that Partitioned Hash-Join significantly improves performance not only for MonetDB and DSM, but also for the NSM pre-projection strategy.

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

As for the prospects of applying DSM Radix-Decluster in off-the-shelf RDBMS products, we support the case made in [30] for systems that combine DSM and NSM natively, or that simply add DSM to the normal NSM representation as *projection indices* [29].

4.4 Modeling Memory Access Costs

Cache-conscious database algorithms 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 [26, 24].

Memory access cost can be modeled by estimating the number of cache misses **M** and scoring them with their respective miss latency l [27]. 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, calculating the total cost as sum of the cost for all levels:

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

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*, such as *sequential* or *random*. 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. The details are presented in [26, 24].

5. VECTORIZED COLUMN-STORES

Our experiments on cache-conscious algorithms revealed the benefits and pitfalls of early versions of MonetDB and triggered projects exploring yet again widely different avenues. One stream focuses on exploring self-adaptive database architecture as discussed in Section 6.1. A second stream aims at interleaving query optimization and query execution at runtime, exploiting sampling over materialized intermediate results to detect data correlations [20]. A third stream forms the X100 project [43] which is geared towards better use of the hardware platforms for columnar execution, while avoiding excessive materialization. [2]

The X100 execution engine at the core of this project conserves the efficient zero-degree of freedom columnar operators found in MonetDB's BAT Algebra, but embeds them in a pipelined relational execution model, where small slices of columns (called "vectors"), rather than entire columns are pulled top-down through a relational operator tree. As such, it cleanly separates columnar data flow from pipelined control flow. The vector size is tuned such that all vectors of a (sub-) query together fit into the CPU cache. When used with a vector-size of one (tuple-at-a-time), X100 performance tends to be as slow as a typical RDBMS, while a size between 100 and 1000 improves performance by two orders of magnitude, providing a proper benchmark for the effectiveness of columnar execution (because measured in the same system).

Rather than relying on memory-mapped files for I/O, X100 uses an explicit buffer manager optimized for sequential I/O, X100 is geared to supporting huge disk-based data-sets efficiently, that is, aiming to keep all CPU cores busy. While the reduced I/O volume in column stores makes non CPU-optimized implementations easily CPU bound, the raw speed of the columnar (vectorized) execution primitives in MonetDB and X100 makes this non-trivial to solve. To reduce I/O bandwidth needs, X100 added vectorized ultra-fast compression methods [44] that decompress values in less than 5 CPU cycles per tuple, as well as the cooperative scan I/O scheduling [45] where multiple active queries cooperate to create synergy rather than competition for I/O resources.

The X100 buffer manager supports both DSM and PAX and decouples storage layout from query execution layout. The execution layout only contains used columns, typically in DSM when passing data sequentially between operators. However, it was shown [46] that (e.g., hash-based) random-access operators benefit from re-grouping columnar data horizontally, creating a mixed-layout in-execution tuple representation, calling for *tuple-layout planning* as a novel query optimizer task.

Current X100 research focuses on achieving fast ways to update disk-based column-stores, as well as the applicability of novel and dynamic multi-table clustering techniques, to reduce further the CPU and I/O effort needed for answering BI queries.

[2] X100 is further developed in the CWI spin-off VectorWise.

6. THE NEW SPECIES

Column-stores, and the traditional n-ary stores are at two ends of a spectrum. They both address the same user community, but with different technological solutions and application emphasis. The space spanned is by far not explored in all its depths. Some species on the horizon based on MonetDB are the following.

6.1 Self-management for Survival

For over a decade [7, 2], reduction of the number of knobs in a database system has been the prime target. The underlying reasons stem both from the software complexity of a modern DBMS and the need to support a broad user base. Users are not necessarily versed in physical database design, index selection, or thorough SQL knowledge.

A column store with its reduced storage complexity, makes it easier for the system designer to strike a balance between software complexity, maintainability, and performance. In this context two directions within the MonetDB project are being explored: *database cracking* and *recycling*.

Since [22], the predominant approach in database systems to pay the price for index maintenance during updates is challenged by a simple scheme, called *database cracking*. The intuition is to focus on a non-ordered table organization, extending a partial index with each query, i.e., the physical data layout is reorganized within the critical path of query processing. We have shown that this approach is competitive over upfront complete table sorting and that its benefits can be maintained under high update load. The approach does not require knobs [18].

Another track concerns an alternative to materialized views, which are produced by design wizards through analysis of the past workload. However, the operator-at-a-time paradigm with full materialization of all intermediates pursued in MonetDB provides a hook for easier materialized view capturing. The results of all relational operators can be maintained in a cache, which is also aware of their dependencies. Then, traditional cache replacement policies can be applied to avoid double work, cherry picking the cache for previously derived results. It has been shown to be effective using the real-life query log of the Skyserver [19].

6.2 Turbulent Data Streams

The abundance of processing power in the network, its capabilities and those of individual nodes has triggered an avalanche of research in the area of P2P systems and the Cloud. The time has come where the network can be your friend in addressing the performance requirements of large databases. The DataCell [21, 23] and DataCyclotron [13] projects represent species in this arena.

The DataCell aims at using the complete software stack of MonetDB to provide a rich data stream management solution. Its salient feature is to focus on incremental bulk-event processing using the binary relational algebra engine. The enhanced SQL functionality allows for general predicate based window processing. Preliminary results indicate the validity of this approach against public benchmarks [23].

Common off-the-shelf network interface hardware also provides opportunities to increase performance. In particular, Remote DMA enables the nodes in a cluster to write into remote memory without interference of the CPU. This completely bypasses the TCP/IP stack, which is known for its high overhead. This hardware feature can be used to explore the survival of a new species, one where the database hot-set is continuously floating around the network. The obvious benefit, if successful, would be increased system throughput and an architecture to exploit the opportunities offered by clusters and compute Clouds.

7. EVOLUTIONARY TRENDS

In retrospect, all three architecture-aware VLDB 1999 papers [6, 31, 9] deserve to be considered the seeds of a renewed research interest in making database technology hardware-conscious to effectively exploit the available performance potentials. Since then, an active community, ranging from database researchers to hardware designers, has established around the DaMoN workshop series [4].

While projects like MonetDB and C-Store [37, 1] demonstrate the natural fitness of columnar systems, various other projects look into making also classical row-store technology (more) hardware aware. We briefly summarize selected work that has been published in major database conferences over the last decade.

As discussed in Section 4, improving spatial and temporal locality of data access is a key technique to cope with increased memory access latency by reducing cache miss ratios. Ailamaki et al. [6] present a detailed problem analysis, identifying that commercial DBMSs exploit modern hardware poorly, causing CPUs to stall for up to 95% of the time waiting for memory.

Rao and Ross focus on tree structures, presenting read-only *Cache-Sensitive Search Trees* (CSS-trees) in [31]. The key ideas are to eliminate internal-node pointers by storing the tree as an array, and to tune the node size to maximize cache line usage. In [32], these ideas are extended to updateable *Cache-Sensitive B^+ Trees* (CSB$^+$-Trees), improving spatial locality of B$^+$-Trees by storing all children of a given node contiguously. Thus, only the pointer of the first child needs to be stored explicitly, reducing the node size and increasing cache line utilization. In [40], Zhou and Ross improve the temporal locality of indices, by buffering accesses to nodes in tree structures, and performing tree-traversal operations in bulk.

Ailamaki et al. [5] design a hybrid data storage model, called PAX. By keeping a NSM-like paged storage, but using a DSM-like columnar layout within each disk page, PAX has the I/O characteristics of NSM, and cache-characteristics of DSM. The *data-morphing* technique [14] generalizes the PAX approach by dynamically dividing a relation into a set of vertical partitions. Finally, Clotho [33] lifts the restriction that storage and processing use the same model, by transparently choosing the most suitable storage model for a given workload. Chen et al. [10] discuss how explicit software prefetching techniques can be applied to database operations to reduce the effect of cache and memory access latency.

Most of the techniques mentioned above need to be explicitly tuned to the hardware characteristics to achieve their optimal performance. In contrast, *cache-oblivious algorithms* [17] and their data structures are designed to maximize spatial temporal locality independently of the exact hardware parameters.

While not noticeable with operator-at-a-time execution paradigms, instruction cache misses have been identified as a major performance problem for tuple-at-a-time execution paradigms [6]. To overcome the problem, various techniques are proposed to improve temporal code locality with tuple-at-a-time execution [15, 41].

Space limitations do not allow us to discuss additional work, e.g., on super-skalar multi-core CPUs, SIMD, graphics processors (GPUs), network processors, FPGAs, and flash memory.

8. SUMMARY

This short note provides a synopsis of the evolution of an open-source column-store DBMS. During its development we hit several sizable areas for innovative research. The memory chasms, between L1/L2, RAM and disk, and processing speed of multi-core systems call for new ways to design algorithms. From complex navigational datastructures back to ingenious simplicity using high speed sequential access combined with partial indexing.

Operator-at-a-time processing, albeit potentially expensive due to its full materialization, can be bent into an advantage by reducing the need to pre-define materialized views. Likewise, the processing paradigm shift turns the page from preparing for ubiquitous fast retrieval using expensive updates into preparing for fast access to what matters only.

Evolution takes time. Existing species adjust to the change in environments. Some species become extinct, others can adjust and flourish in the new setting. The prime shift of database transaction processing towards business intelligence applications in a turbulent hardware setting, can be considered such a major environmental shift. It opens a vista for exploration for the brave at heart. Pioneers have spotted a lot of game awaiting for the hunting researchers.

Acknowledgments. *We wish to thank all (former) members of the CWI database group for their effort, inspiration and dedication to make MonetDB a success. Without them and our responsive, diligent users we would have been left standing empty handed.*

9. REFERENCES

[1] D. J. Abadi. *Query Execution in Column-Oriented Database Systems*. PhD thesis, MIT, Cambridge, MA, USA, 2008.

[2] R. Agrawal, A. Ailamaki, P. A. Bernstein, E. A. Brewer, M. J. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. J. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. F. Korth, D. Kossmann, S. Madden, R. Magoulas, B. C. Ooi, T. O'Reilly, R. Ramakrishnan, S. Sarawagi, M. Stonebraker, A. S. Szalay, and G. Weikum. The claremont report on database research. *Communications of the ACM*, 52(6):56–65, 2009.

[3] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. arasayya, and M. Syamala. Database Tuning Advisor for Microsoft SQL Server. In *VLDB*, 2004.

[4] A. Ailamaki, P. A. Boncz, S. Manegold, Q. Luo, and K. A. Ross, editors. *Int'l Workshop on Data Management on New Hardware (DaMoN)*, 2005–2009. http://dblp.uni-trier.de/db/conf/damon/.

[5] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, 2001.

[6] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where does time go? In *VLDB*, 1999.

[7] P. A. Bernstein, M. L. Brodie, S. Ceri, D. J. DeWitt, M. J. Franklin, H. Garcia-Molina, J. Gray, G. Held, J. M. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. F. Naughton, H. Pirahesh, M. Stonebraker, and J. D. Ullman. The asilomar report on database research. *ACM SIGMOD Record*, 27(4):74–80, 1998.

[8] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *SIGMOD*, 2006.

[9] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*, 1999.

[10] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)*, 32(3), 2007.

[11] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *SIGMOD*, 1985.

[12] R. Cornacchia, S. Heman, M. Zukowski, A. P. de Vries, and P. A. Boncz. Flexible and efficient IR using Array Databases. *The VLDB Journal*, 17(1):151–168, Jan. 2008.

[13] P. Frey, R. Goncalves, M. L. Kersten, and J. Teubner. Spinning Relations: High-Speed Networks for Distributed Join Processing. In *DaMoN*, 2009.

[14] R. A. Hankins and J. M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB*, 2003.

[15] S. Harizopoulos and A. Ailamaki. STEPS towards Cache-resident Transaction Processing. In *VLDB*, 2004.

[16] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: a simultaneously pipelined relational query engine. In *SIGMOD*, 2005.

[17] B. He and Q. Luo. Cache-oblivious query processing. In *CIDR*,2007.

[18] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column-stores. In *SIGMOD*, 2009.

[19] M. Ivanova, M. L. Kersten, N. Nes, and R. Goncalves. An Architecture for Recycling Intermediates in a Column-store. In *SIGMOD*, 2009.

[20] R. A. Kader, P. A. Boncz, S. Manegold, and M. van Keulen. ROX: Run-time Optimization of XQueries. In *SIGMOD*, 2009.

[21] M. L. Kersten, E. Liarou, and R. Goncalves. A Query Language for a Data Refinery Cell. In *Int'l Workshop on Event-driven Architecture, Processing and Systems (EDA-PS)*, 2007.

[22] M. L. Kersten and S. Manegold. Cracking the Database Store. In *CIDR*, 2005.

[23] E. Liarou, R. Goncalves, and S. Idreos. Exploiting the Power of Relational Databases for Efficient Stream Processing. In *EDBT*, 2009.

[24] S. Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance*. PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, Dec. 2002.

[25] S. Manegold, P. A. Boncz, and M. L. Kersten. What happens during a Join? — Dissecting CPU and Memory Optimization Effects. In *VLDB*, pages 339–350, Cairo, Egypt, Sept. 2000.

[26] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *VLDB*, 2002.

[27] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Main-Memory Join On Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(4):709–730, 2002.

[28] S. Manegold, P. A. Boncz, N. Nes, and M. L. Kersten. Cache-Conscious Radix-Decluster Projections. In *VLDB*, 2004.

[29] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *SIGMOD*, 1997.

[30] R. Ramamurthy, D. DeWitt, and Q. Su. A Case for Fractured Mirrors. In *VLDB*, 2002.

[31] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB*, 1999.

[32] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *SIGMOD*, 2000.

[33] M. Shao, J. Schindler, S. W. Schlosser, A. Ailamaki, and G. R. Ganger. Clotho: decoupling memory page layout from storage organization. In *VLDB*, 2004.

[34] A. Shatdal, C. Kant, and J. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *VLDB*, 1994.

[35] A. Shoshani and D. Rotem. *Scientific Data Management Challenges, Technology, and Deployment*. Chapman & Hall, 2009.

[36] L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. Column-Store Support for RDF Data Management: not all swans are white. In *VLDB*, 2008.

[37] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-Oriented DBMS. In *VLDB*, 2005.

[38] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*, 2007.

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

[40] J. Zhou and K. A. Ross. Buffering Accesses to Memory-Resident Index Structures. In *VLDB*, 2003.

[41] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *SIGMOD*, 2004.

[42] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, 2004.

[43] M. Zukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. PhD thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, Sept. 2009.

[44] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, 2006.

[45] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB*, 2007.

[46] M. Zukowski, N. Nes, and P. A. Boncz. DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing. In *DaMoN*, 2008.