Chapter 8

Conclusion

8.1 Contributions

Let us recollect the main research questions defined for this thesis in Section 3.1:

- how to get best performance out of modern CPU and memory hardware on query-intensive DBMS applications?
- how to support multiple (complex) data models?
- how to provide sufficient extensibility to new domains?

The contributions of this thesis to answering these research questions are summarized below.

The MIL Language The MIL language and its BAT algebra in particular provide a clean and powerful query language that is extensible in all its dimensions. By focusing on column-at-a-time operators, rather than a higher level abstraction of e.g. tables or objects, the language is neutral to various data models. This thesis describes in detail how MIL can be used in a relational DBMS and also gives examples of use in object-oriented query processing. In various collaboration projects, MIL has successfully been used for query processing on networked (bayesian belief networks) [dVW98], topological (triangulated interconnection networks) [WvZF+98] and semi-structured (XML) [SKWW00] data models as well. Note that these data models store data not in blobs or even ADTs [RS97], but decompose their data in BATs such that e.g. neighbor predicates can be tested by MIL set-operators. Thus, there are indeed strong indications that MIL fulfills its goal of being the building block for database systems with widely different data models.

Moreover, the column-at-a-time nature of the BAT algebra yields operators with a low degree of freedom, making MIL versus relational algebra in query processing what RISC is to CISC in CPU technology. This "RISC approach" enables optimization techniques in the MIL implementation for squeezing best performance out of modern super-scalar CPUs, which was also one of the stated goals.
The Monet System  The benchmarking of Monet on O07 [BKK96], Sequoia [BQK96], TPC-D [BWK98] as well as the Drill Down Benchmark [BRK98] provides proof for the feasibility of efficient query processing with MIL.

The Monet system uses a coherent mix of design and implementation ideas (database architecture) that provides useful insight for future database kernel programmers. Techniques like implicit storage in void columns, view-implementations and type remappings were effective in eliminating the extra join-overhead that is encountered when relational or object-oriented applications are fully decomposed into the binary table model. Main-memory optimization methods like code expansions were employed throughout the system to make its code as efficient as scientific computation code like matrix multiplication can be, which is otherwise far out of reach for DBMS software [BGB98, KPH+98, TLPZT97, ADHW99].

While this thesis focuses on the core issues concerning Monet, during the course of this research, Monet has successfully been applied in commercial and scientific projects in a wide range of application areas. The author personally participated in the creation of the data mining spin-off company Data Distilleries during the course of this Ph.D. track, and later as full-time overall system architect. Data Distilleries is now a global player in the analytical CRM market, and still uses Monet as back-end in its analytical tool. This tool combines interactive, multi-algorithm, data mining functionality with ad-hoc OLAP. Achieving more than 36-fold speed-up over Oracle8 in data mining loads [BRK98], and running repositories larger than 100GB in daily production at the data centers of the largest Dutch financial institutions, Monet has proven its value in real-life applications.

Monet is also used for scientific applications in the areas of GIS [BK95, BQK96, WvZF+98], XML [SWK+01, SKW01, WSK99], information retrieval [dVB98] and multi-media [NK97, NQK98, NK98, KNW98, dV99, Nes00, dV00, dVWAK00, BdVNK01, dVMNK02]. This has led to a wealth of extension modules that introduce new atomic types, search accelerators and algebraic operators to MIL.

In the near future, our research group will make Monet and an SQL-2 front-end as described in this thesis publicly available in open source.

Run-time Query Optimization  Monet defers important physical query optimization decisions to run-time, both enhancing their quality and simplifying the query optimization process. What makes MIL stand out from other database languages is that it is both a logical and an execution language. By providing a direct implementation for this logical algebra language, the Monet system separates query optimization in a strategical and a tactical phase. Query optimizing systems producing MIL code must transform a high-level query in a sequence of appropriate MIL primitives. This includes determining a good (join-)order, but excludes translation to physical primitives. Choosing an algorithm is performed at run-time inside each MIL operator during the process of tactical query optimization. By using properties maintained on relation fragments and full propagation of these properties across operators, Monet conserves maximum information about the data that is being processed, which is also combined with run-time system statistics. By off-loading these activities from the compile-time phase, this approach simplifies the task of query optimizers, and has the potential to take better optimization decisions, as more run-time information is taken into account.
8.1. CONTRIBUTIONS

**Cache-Conscious Query Processing**  New Radix-Algorithms that provide the basic building blocks for cache-optimized query processing, were contributed in this thesis. These algorithms trade memory access cost for extra CPU cost. Given the fact that memory latency does not improve (significantly) but CPU power increases exponentially each year, this has become a highly beneficial trade-off in the past decade, and will even be more so in the next. We have shown on hardware from 1999 that the Radix-Algorithms can accelerate equi-join by almost a magnitude (while CPUs from 2002 are three times more powerful yet memory latency has improved nothing). Thus, we conclude that the "battleground" for database query optimization is shifting from I/O to the DRAM layer in the memory hierarchy.

As CPU caches are even more uncontrollable than virtual memory, it is not possible to create an explicit "CPU-cache-buffer-manager" in the DBMS. Rather, one must just rely on the hardware-implemented eviction mechanism (usually LRU). The implicit idea behind Partitioned Hash-Join and also our Radix-Algorithms is to let it work well on a LRU cache. An interesting property of such algorithms is that when it runs well on one cache level, it also runs well on the lower levels (but not the other way around). However, virtual memory is also an LRU cache (fully associative and with large granularity and latency). Thus, when one switches to LRU-cache optimized query processing algorithms with the initial purpose of optimizing CPU cache utilization, the access patterns may become so predictable that a DBMS buffer manager is not strictly needed anymore!

In the case of cache-conscious query processing in Monet, extra performance is gained by the vertical decomposition and data compression (enumeration types) as these optimize cache-line usage. The column-wise query processing is elegantly made cache-conscious by the cluster-decluster join strategy, based on the innovative Radix-Decluster algorithm, which represents a significant new research result. The discussion of the SQL-2 front-end showed that by using this strategy, all MIL code generated for the relational algebra operators can be cache-optimized. Selections are sequential, equi-joins use the Radix-Partitioned Hash-Join, ORDER BYS rely on Quick-Sort and the cluster-decluster join strategy, while GROUP BYS make a dynamic choice depending on the cache size between hash-grouping or ORDER BY.

**Explicit Transaction Management**  A novel idea tested out in Monet is to construct a query language that in itself does not provide ACID properties, rather provides the building blocks for creating a failsafe transaction system. The advantage of this approach is that queries need only pay for the overheads of transaction management if their application requires transaction functionality. Many query-intensive applications work with bulk-updates or are append-only, so it is a waste to give up precious query performance in e.g. a complicated ARIES algorithm that is intertwined with the buffer manager and random I/O constraints [MHL+92]. While it is easy to point out that absence of transaction management may enhance performance, this thesis also tries to make clear that by providing a carefully defined set of update primitives, MIL is capable of supporting sophisticated transaction functionality.


8.2 Future Work

While we consider Monet a largely successful project, there are a many areas were improvements are possible:

**Tactical Query Optimization Language Issues** Currently, Monet maintains a hard-coded (non-extensible) set of BAT- and column-properties inside the BAT descriptor, whose propagation by the various MIL operators is also hard-coded in their operator implementations. This conflicts with the extensibility requirement (new domains might need new properties) and is error-prone.

Another issue is the fact that some MIL operators still make tactical decisions inside their C implementations. It can be considered just an implementation issue to convert this decision code into proper scripted MIL procedures. However, doing so will expose a large collection of MIL physical operators that clutter the language, such that it is an interesting question whether MIL should, like other query algebras [Güt89], go back to a design with a separate logical and a physical algebra level.

In any case, it would be much more elegant if MIL operator signatures could be extended with predicates that go beyond the types of the parameters and e.g. contain predicates or even rule-expressions on BAT- and column-properties.

Finally, the propagation of such properties would better be described explicitly in propagation rules. It is an open question how powerful such a rule system would be, and whether it would take over the role of MIL procedures as the vehicle for tactical optimization entirely. Also, the interaction of such a rule system with query rewrite systems, such as the MIL squeezer [MPK00] that performs multi-query optimizations, should be studied further.

**Pure Column Algebra** In the discussion of query processing strategies, we almost exclusively use BATs that have one void column, as such BATs can be joined together at trivial cost. Conversely, not using such BATs can seriously hinder performance. The question then rises whether a "true" single-column algebra would not even be more elegant. This would only really cause problems for the MIL-join and -order, as these are currently the only operators to produce "real" two-column BATs.

Thus, this column algebra would need to allow an operator to deliver multiple columns as a result. Giving back multiple results is also a functional wish we encountered e.g. in multi-media retrieval. A problem with multiple results is that introducing a tuple or record type should be avoided in order to keep the language a clean column-oriented algebra and thus data model neutral. A possible solution would be to drop the concept of nested functional-style expressions, going back to an assembly-like language where programs consist of sequences of single-operator statements that fill one or multiple variables.

**Programmable Virtual Memory On RAID** One of our conclusions is that the increasing memory access bottleneck could spell the end of the DBMS buffer manager, handling I/O implicitly via virtual memory. One of the issues that needs investigation is whether current virtual memory implementations can indeed support such an architecture, and in particular whether it can sustain sequential or quasi-sequential LRU cacheable access patterns (such as bi-directional scans of Quick-Sort, or the multi-cursor sequential scan from Radix-Cluster) at sufficiently high bandwidth in order to
8.2. FUTURE WORK

not let the query processing operators be I/O limited. A rough rule-of-thumb is that the typical MIL operator uses in the range of 40-100 CPU cycles per tuple, thus a 1GHz CPU consumes 10-25 million tuples per second, which equals 40-100MB/s (assuming 4-byte wide BUNs).

Such bandwidths are attainable by (cheap) RAID devices (or software RAID) but the question is whether the OS implementations of virtual memory are up to this. Our early indications are that performance should be improved, and this could be done using virtual memory reprogramming techniques previously used in OODBMSs [LOW91, Ple97] by reprogramming catching page faults and scheduling large asynchronous I/O read-ahead operations according to specified access pattern hints.

Parallel Execution As hardware trends continue as now, the introduction of Simultaneous Multi-Threading (SMT) and multi-processing on-a-chip will create a situation that all commodity processor chips contain multiple (logical) CPUs, making the pervasiveness of parallel processing finally a reality. At the same time, it is obvious that highly scalable systems use a shared-nothing architecture, thus the future of parallel processing architectures is hybrid.

At the start of this research, we had hoped to pay more attention to investigating the interaction between column-wise query processing and hybridly parallel machine architectures. This remains an interesting research area to be pursued in the future. In this context, it would be particularly interesting to look at cache-conscious parallel query processing, as the Radix-Algorithms presented in this thesis are designed for uniprocessor execution only.

Just-In-Time Query Compilation Monet can make use of highly optimized implementations of its algebraic operators, as these work column-wise and have a low degree of freedom (i.e. they are “RISCy”).

A different road to obtain high CPU efficiency would be to use just-in-time query compilation with real-time code-expansion. Such a system architecture would parse queries and generate highly CPU-efficient Java/C/C++-code, possibly even enriched by pieces of machine-specific assembly (e.g. containing SSE2 instructions). That code is then natively compiled into assembly on-the-fly, and loaded as a shared library, and kept around for re-use later on. The data model to be used would – given our good experiences – use vertically fragmented structures, but it is an open question what query language would best be suited for such a system. The idea might be pursued in MIL, possibly compiling multiple nested MIL operators into one execution operator, but it is possible, that better results would be obtained by doing this on an tuple- or object-algebra as these inherently have more work available per tuple.