MonetDB: Two Decades of Research in Column-oriented Database Architectures

Stratos Idreos Fabian Groffen Niels Nes Stefan Manegold Sjoerd Mullender Martin Kersten

Database Architectures group*, CWI, Amsterdam, The Netherlands

Abstract

MonetDB is a state-of-the-art open-source column-store database management system targeting applications in need for analytics over large collections of data. MonetDB is actively used nowadays in health care, in telecommunications as well as in scientific databases and in data management research, accumulating on average more than 10,000 downloads on a monthly basis. This paper gives a brief overview of the MonetDB technology as it developed over the past two decades and the main research highlights which drive the current MonetDB design and form the basis for its future evolution.

1 Introduction

MonetDB[1] is an open-source database management system (DBMS) for high-performance applications in data mining, business intelligence, OLAP, scientific databases, XML Query, text and multimedia retrieval, that is being developed at the CWI database architectures research group since 1993 [19].

MonetDB was designed primarily for data warehouse applications. These applications are characterized by large databases, which are mostly queried to provide business intelligence or decision support. Similar applications also appear frequently in the area of e-science, where observations are collected into a warehouse for subsequent scientific analysis. Nowadays, MonetDB is actively used in businesses such as health care and telecommunications as well as in sciences such as in astronomy. It is also actively used in data management research and education. The system is downloaded more than 10,000 times every month.

MonetDB achieves significant speed up compared to more traditional designs by innovations at all layers of a DBMS, e.g., a storage model based on vertical fragmentation (column-store), a modern CPU-tuned query execution architecture, adaptive indexing, run-time query optimization, and a modular software architecture. The rest of the paper gives a brief overview of the main design points and research directions.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Acknowledgments. The research and development reported here has been made possible by all former and current members of CWI's Database Architectures group, most notably Peter Boncz, Romulo Goncalves, Sandor Heman, Milena Ivanova, Erietta Liarou, Lefteris Sidirourgos, Ying Zhang, Marcin Zukowski.

[1] http://www.monetdb.org/

2 MonetDB Design

From a user's point of view, MonetDB is a full-fledged relational DBMS that supports the SQL:2003 standard and provides standard client interfaces such as ODBC and JDBC, as well as application programming interfaces for various programming languages including C, Python, Java, Ruby, Perl, and PHP.

MonetDB is designed to exploit the large main memories of modern computer systems effectively and efficiently during query processing, while the database is persistently stored on disk. With respect to performance, MonetDB mainly focuses on analytical and scientific workloads that are read-dominated and where updates mostly consist of appending new data to the database in large chucks at a time. However, MonetDB also provides complete support for transactions in compliance with the SQL:2003 standard.

Internally, the design, architecture and implementation of MonetDB reconsiders all aspects and components of classical database architecture and technology by effectively exploiting the potentials of modern hardware. MonetDB is one of the first publicly available DBMSs designed to exploit column-store technology. MonetDB does not only use the column-oriented logic for the way it stores data; it provides a whole new design for an execution engine that is fully tailored for columnar execution, deploying carefully designed cache-conscious data structures and algorithms that make optimal use of hierarchical memory systems [2].

In addition, MonetDB provides novel techniques for efficient support of a priori unknown or rapidly changing workloads over large data volumes. Both the fine-grained flexible intermediate result caching technique "recycling" [12] and the adaptive incremental indexing technique "database cracking" [8] require minimal overhead and investment to provide maximal benefit for the actual workload and the actual hot data.

The design also supports extensibility of the whole system at various levels. Via extension modules, implemented in C or MonetDB's MAL language, new data types and new algorithms can be added to the system to support special application requirements that go beyond the SQL standard, or enable efficient exploitation of domain-specific data characteristics. Additionally, MonetDB provides a modular multi-tier query optimization framework that can be extended with domain specific optimizer rules.

Finally, the core architecture of MonetDB has proved to provide efficient support not only for the relational data model and SQL, but also for, e.g., XML and XQuery [1]. In this line, support for RDF and SPARQL, as well as arrays [14] is currently under development.

Physical Data Model. The storage model is a significant deviation of traditional database systems. Instead of storing all attributes of each relational tuple together in one record (aka. *row-store*), MonetDB represents relational tables using vertical fragmentation (aka. *column-store*), by storing each column in a separate `(surrogate,value)` table, called a *BAT* (*Binary Association Table*). The left column, often the surrogate or `OID` (object-identifier), is called the *head*, and the right column, usually holding the actual attribute values, is called the *tail*. In this way, every relational table is internally represented as a collection of BATs. For a relation R of k attributes, there exist k BATs, each BAT storing the respective attribute as (`OID`,`value`) pairs. The system-generated `OID` identifies the relational tuple that the attribute `value` belongs to, i.e., all attribute values of a single tuple are assigned the same `OID`. `OID` values form a dense ascending sequence representing the *position* of a value in the column. Thus, for base BATs, the `OID` columns are not materialized, but rather implicitly given by the position. This makes base BATs essentially equal to typed arrays in C with optional metadata. For each relational tuple t of R, all attributes of t are stored in the *same* position in their respective column representations. The position is determined by the insertion order of the tuples. This tuple-order *alignment* across all base columns allows the column-oriented system to perform tuple reconstructions efficiently in the presence of tuple order-preserving operators.

For fixed-width data types (e.g., integer, decimal and floating point numbers), MonetDB uses a plain C array of the respective type to store the `value` column of a BAT. For variable-width data types (e.g., strings), MonetDB applies a kind of dictionary encoding. All distinct values of a column are stored in a BLOB and the `value` column of the BAT is an integer array with references to BLOB positions where the actual values exist.

MonetDB uses the operating system's memory mapped files support to load data in main memory and

exploit extended virtual memory. Thus, all data structures are represented in the same binary format on disk and in memory. Furthermore, MonetDB uses late tuple reconstruction, i.e., during the entire query evaluation all intermediate results are in a column format. Only just before sending the final result to the client, N-ary tuples are constructed. This approach allows the query engine to exploit CPU- and cache-optimized vector-like operator implementations throughout the whole query evaluation relying on a bulk processing model as opposed to the typical Volcano approach, allowing to minimize function calls, type casting, various metadata handling costs, etc. Intermediate results need to be materialized, but those can be reused [12].

Execution Model. The MonetDB kernel is an abstract machine, programmed in the *MonetDB Assembly Language* (*MAL*). The core of MAL is formed by a closed low-level two-column relational algebra on BATs. N-ary relational algebra plans are translated into two-column BAT algebra and compiled to MAL programs. These MAL programs are then evaluated in a *operator-at-a-time* manner, i.e., each operation is evaluated to completion over its entire input data, before subsequent data-dependent operations are invoked. Each BAT algebra operator maps to a simple MAL instruction, which has zero degrees of freedom in its behavior (obviously, it may be parameterized where necessary): it does not take complex expressions as parameter. Complex operations 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 avoid an expression interpreting engine; rather, all BAT algebra operations in the implementation map onto simple array operations. The figure on the left shows such an implementation of a select operator. The BAT algebra operators have the advantage that tight for-loops without function calls 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.

```
select(B:bat[:oid,:int], V:int) ⟶ for (i = j = 0; i < n; i++)
                                    if (B.tail[i] == V)
                                      R.tail[j++] = i;
```

System Architecture. MonetDB's query processing scheme is centered around three software layers.

Front-end. The top layer or *front-end* provides the user-level data model and query language. While relational tables with SQL and XML with XQuery [1] are readily available, arrays with SciQL [14] and RDF with SPARQL are on our research and development agenda. The front-end's task is to map the user-space data model to MonetDB's BATs and to translate the user-space query language to MAL. The query language is first parsed into an internal representation (e.g., SQL into relational algebra), which is then optimized using domain-specific rules. In general, these domain-specific *strategic optimizations* aim primarily at reducing the amount of data to be processed, i.e., the size of intermediate results. In the case of SQL & relational algebra, such optimizations include heuristics like pushing down selections and exploiting join indexes. The optimized logical plan is then translated into MAL and handed over to the back-end for general MAL-optimization and evaluation.

Back-end. The middle layer or *back-end* consists of the MAL optimizers framework and the MAL interpreter as textual interface to the kernel. The MAL optimizers framework consists of a collection of optimizer modules that each transform a given MAL program into a more efficient one, possibly adding resource management directives. The modules provide facilities ranging from symbolic processing up to just-in-time data distribution and execution. This *tactical optimization* is more inspired by programming language optimization than by classical database query optimization. It breaks with the hitherto omnipresent cost-based optimizers, recognizing 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.

Kernel. The bottom layer or *kernel* (aka. *GDK*) provides BATs as MonetDB's bread-and-butter data structure, as well as the library of highly optimized implementations of the binary relational algebra operators. Due to the operator-at-a-time "bulk-processing" evaluation paradigm, each operator has access to its entire input including known properties. This allows the algebra operators to perform *operational optimization*, i.e., to choose at runtime the actual algorithm and implementation to be used, based on the input's properties. For instance, a `select` operator can exploit sorted-ness of a BAT by deploying binary search, or (for point-selections) use an existing hash-index, and fall back to a scan otherwise. Likewise, a join can at runtime decide to, e.g., perform a merge-join if the join attributes happen to be sorted, and fall-back to hash-join otherwise.

3 Research

In this section, we briefly summarize the highlights of column-oriented research in the context of MonetDB. Part of the topics discussed below reflect fundamental research in database architectures which has lead to the current design of MonetDB and its spin-offs. Another part of the research topics reflects high risk research aiming at future innovations in database architectures and big data analytics. Both fundamental and high risk projects are fully materialized within the MonetDB kernel and are disseminated as open source code together with the rest of the MonetDB code family.

Hardware-conscious Database Technology. A key innovation in MonetDB is its reliance on hardware conscious algorithms. 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, including database management systems; a phenomenon widely known as the "memory wall".

The crucial aspect in order to overcome the memory wall is good use of CPU caches, i.e., careful tuning of memory access patterns is needed. This led to a new breed of query processing algorithms, in particular for join processing, such as *partitioned hash-join* [3] and *radix-cluster* [18]. The key idea is to restrict any random data access pattern to data regions that fit into the CPU caches to avoid cache misses, and thus, performance degradation. For query optimization to work in a cache-conscious environment, and to enable automatic tuning of our cache-conscious algorithms on different types of hardware, we developed a methodology for creating cost models that take the cost of memory access into account [17]. The key 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*.

Vectorized Execution and Light-weight Compression. The X100 project explored a compromise between classical tuple-at-a-time pipelining and operator-at-a-time bulk processing [4]. The idea of vectorized execution is to operate on chunks (vectors) of data that are large enough to amortize function call overheads, but small enough to fit in CPU caches and to avoid materialization of large intermediates into main memory. Combined with just-in-time light-weight compression, it lowers the memory wall significantly [21]. The X100 project has been commercialized into the Actian/VectorWise company.

Reusing Intermediate Results with Recycler. Bulk processing in a column-store architecture implies materialization of intermediate results. The *Recycler* project adaptively stores and reuses those intermediates when possible, i.e., when a select operator is covered by a stored intermediate of a past query, then MonetDB avoids touching the base column [**?**]. Intermediates are kept around as long as they fit in the allocated space for the Recycler and as long as they are hot.

Adaptive Indexing and Database Cracking. Modern business applications and scientific databases call for inherently dynamic data storage environments. Such environments are characterized by two challenging features: (a) they have little idle system time to devote on physical design; and (b) there is little, if any, a priori workload knowledge, while the query and data workload keeps changing dynamically. In such environments, traditional approaches to index building and maintenance cannot apply.

To solve this problem, MonetDB research pioneered *Database cracking* that allows on-the-fly physical data reorganization, as a collateral effect of query processing [8]. Cracking continuously and automatically adapts indexes to the workload at hand, without human intervention. Indexes are built incrementally, adaptively, and on demand as part of select operators, join operators and projection operators; the more queries are processed the more the relevant indexes are optimized.

Cracking was proposed in the context of modern column-stores and has been hitherto applied for boosting the select operator performance [8], maintenance under updates [9], and arbitrary multi-attribute queries with *sideways cracking* [10]. With *partial cracking*, a storage threshold can be applied and cracking adaptively maintains its auxiliary structures within this threshold [10]. A new variant, *stochastic cracking* maintains and expands the adaptive behavior across various workloads by opportunistically introducing random index refinement optimization actions during query processing [7]. Furthermore, contrary to first impressions, database cracking

allows for queries to run concurrently even though at the conceptual level it turns read queries into write queries. With careful and adaptive management of short term latches and with early release of those latches multiple queries can operate in parallel over the cracking indexes [6]. In addition, more recently, database cracking has been extended to exploit a partition/merge -like logic as well as to study the various tradeoffs of adaptive indexing in more detail [11]. When to optimize an adaptive index and by how much are fundamental questions that this research tries to answer.

DataCyclotron. One of the grand challenges of distributed query processing is to devise a self-organizing architecture which exploits all hardware resources optimally to manage the database hot-set, to minimize query response time, and to maximize throughput without single point global co-ordination. The Data Cyclotron architecture [5] addresses this challenge using turbulent data movement through a storage ring built from distributed main memory and capitalizing on the functionality offered by modern remote-DMA network facilities. Queries assigned to individual nodes interact with the storage ring by picking up data fragments that are continuously flowing around, i.e., the hot-set.

Adaptive Sampling and Exploration with SciBORQ. In modern applications, not all data is equally useful all the time. A strong aspect in query processing is exploration. In the SciBORQ project, we explore a route based on exactly this knowledge that only a small fraction of the data is of real value for any specific task [20]. This fraction becomes the focus of scientific reflection through an iterative process of ad-hoc query refinement. However, querying a multi-terabyte database requires a sizeable computing cluster, while ideally the initial investigation should run on the scientist's laptop. We work on strategies on how to make biased *snapshots* of a science warehouse such that data exploration can be instigated using precise control over all resources.

MonetDB/DataCell: Stream Processing in a Column-store. New applications are increasingly transformed into online and continuous streaming applications, e.g., new data continuously arrives, and we need to do analytics in a small time window until the next data batch arrives. This scenario occurs in network and big clusters monitoring, in web logs, in the financial market and in many more applications. In the DataCell project, we design a stream engine on top of MonetDB [16] with a target to allow for advanced analytics "as the data arrives". The major challenge is the efficient support for specialized stream processing features such as window based processing and fast responses as data arrive as well as support for advanced database features such as indexing over the continuously changing data stream.

Graph Databases and Run Time Optimization. Optimization of complex XQueries combining many XPath steps and joins is currently hindered by the absence of good cardinality estimation and cost models for XQuery. Additionally, the state of the art of even relational query optimization still struggles to cope with cost model estimation errors that increase with plan size, as well as with the effect of correlated joins and selections. With ROX, we propose to radically depart from the traditional path of separating the query compilation and query execution phases, by having the optimizer execute, materialize partial results, and use sampling based estimation techniques to observe the characteristics of intermediates [13]. While run-time optimization with sampling removes many of the vulnerabilities of classical optimizers, it brings its own challenges with respect to keeping resource usage under control, both with respect to the materialization of intermediates, as well as the cost of plan exploration using sampling.

4 Deployment

Being freely available in open source, only a fraction of the MonetDB users provide us with detailed information about their deployment of and experiences with MonetDB. In academic environments, MonetDB is used for education and numerous research projects. We use MonetDB in several projects ranging from emergency management (EMILI: `http://emili-project.eu/`) over earth observation (TELEIOS: `http://earthobservatory.eu/`) to astronomy (LOFAR: `http://lofar.org/`) with a general focus on scientific databases (SciLens: `http://scilens.org`), including an implementation of the Sloan Digital Sky

Survey's Data Release 7 (`http://www.scilens.org/skyserverdemo/introduction`). Known commercial deployments range from off-line data analysis for business consultants on 32-bit Windows laptops to large-scale call-detail-record management for telecommunication operators on a farm of large Linux servers.

5 Future Paths

The MonetDB system and continuous research mainly targets the arena of data intensive applications over massive amounts of data such as in scientific databases and the need for analytics in modern businesses. Our future efforts are mainly towards distributed and highly parallel processing as well as towards adaptive and exploratory processing where database systems may *interpret queries by their intent*, rather than as a contract carved in stone for complete and correct answers [15].

References

[1] P. 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.

[2] P. Boncz, M.L. Kersten, and S. Manegold. Breaking the Memory Wall in MonetDB. *Communications of the ACM (CACM)*, 51(12), Dec. 2008.

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

[4] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[5] R. Goncalves and M.L. Kersten. The Data Cyclotron Query Processing Scheme. In *EDBT*, 2010.

[6] G. Graefe, F. Halim, S. Idreos, H. Kuno, and S. Manegold. Concurrency control for adaptive indexing. *PVLDB*, 2012.

[7] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 2012.

[8] S. Idreos, M.L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.

[9] S. Idreos, M.L. Kersten, and S. Manegold. Updating a Cracked Database. In *SIGMOD*, 2007.

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

[11] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9), 2011.

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

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

[14] M.L. Kersten, Y. Zhang, M. Ivanova, and N. Nes. SciQL, a query language for science applications. In *EDBT Workshop on Array Databases*, 2011.

[15] M.L. Kersten, S. Idreos, S. Manegold, and E. Liarou. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. *PVLDB*, 4(12), 2011.

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

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

[18] S. Manegold, P. 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.

[19] S. Manegold, M.L. Kersten, and P. Boncz. Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct. *PVLDB*, 2(2), 2009.

[20] L. Sidirourgos, M.L. Kersten, and P. Boncz. SciBORQ: Scientific data management with Bounds On Runtime and Quality. In *CIDR*, 2011.

[21] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.