# MonetDB And The Application For IR Searches

Weixiong Rao
Department of Computer Science
University of Helsinki, Finland
{wrao}@cs.helsinki.fi

**Abstract**—This seminar report summarizes the design of column store in MonetDB and its application for information retrieval (IR) searches. It covers the design motivation, main components, comparison between MonetDB and other related works, and discussions of potential extension or weakness.

**Keywords**—Column Store, Query Processing, Information Retrieval

## 1 Introduction

MonetDB is an open source relational database management system designed for data warehouse applications. The high performance of MonetDB lies in its innovations of all layers, which are designed to meet the following changes. That is, (i) traditional relational database management systems (RDBMS) were designed for the online transaction processing (OLTP) applications. Such OLTP applications frequently involve a few rows and many columns of relational tables. Nowadays in database applications, besides the OLTP applications, the online analytical processing (OLAP) applications typically process the multi-dimensional cubes involving several columns but a large number of rows of relational tables. (ii) The original assumption of RDBMS was that the disk I/O was treated as the dominating performance factor. Instead modern hardware has become orders of magnitude faster but also orders of magnitude more complex.

The above changes, including the underlying physical hardware and upper application model, lead to the fundamental innovations [3], [7] of the MonetDB architecture, which are summarized as follows.

- Column Store: Traditional RDBMS favors a row-wise fashion for single record lookups. Instead, MonetDB uses column store for analytical queries of large amount of data by better using CPU cache lines.
- Bulk query algebra: Monent designs simplified algebra for much faster implementation on modern hardware.
- Cache-conscious algorithms: due to the limit of memory size, MonetDB carefully uses the CPU caches for tuning of memory access patterns and thus designs a new query processing algorithm (i.e., radix-partitioned hash join).
- Memory access cost modeling: given the new cache-conscious environment, MonetDB develops a new cost model for query optimization. In order to work on diverse modern architecture, the cos model can be parameterized at runtime using automatic calibration techniques.

Since the seminar is about the column databases, this report therefore focuses on the design of column store on MonetDB (and skip the innovations designed for modern hardware), and introduce an application of MonetDB for Information Retrieval (IR) searches [5].

The rest of this report is organized as follows. First Section 2 introduces the design of the column store on MonetDB and then Section 3 gives the application for IR searches. After that, Section 4 reviews related works, and Section 5 finally concludes the report.

## 2 Design of Column Store on MonetDB

### 2.1 Physical Data Model

Different from traditional database systems, MonetDB does not store all attributes of each relational tuple (together in one record), and instead treats a relational table as vertical fragmentations. Thus, MonetDB stores each column of the table in a separate (surrogate,value) table, called a BAT (Binary Association Table). The left column, called head column, is surrogate or OID (object-identier), and only the right column stores the actual attribute values (called tail). As a result, a relation table consisting of $k$ attributes then is represented by $k$ BATs. With the help of the system generated OID, MonetDB needs to lookup the $k$ BATs in order to reconstruct a tuple. In order to perform tuple reconstructions from the $k$ BATs, MonetDB adopts a tuple-order alignment across all base columns. That is, each attribute value belonging to a tuple $t$ is stored in the same position of the associated BAT.

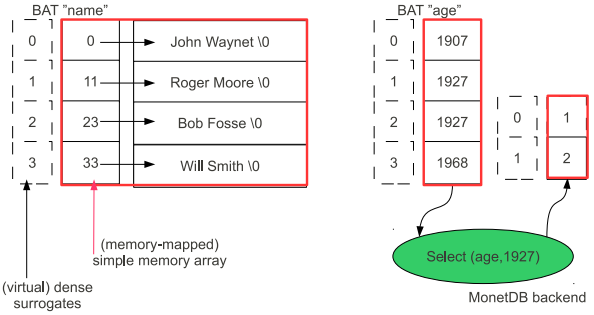


Fig. 1. Colum Store of MonetDB [3]

Next, to represent the tail column, MonetDB considers two cases. (i) For fixed-width data type (e.g., integer, decimal and oating point numbers), MonetDB uses a C-type array. (ii) For

variable-width data types (e.g., strings), MonetDB adopts a dictionary encoding where the distinct values are then store in Blob and the BAT only stores an integer pointer to the Blob position. The BATs "name" and "age"' Fig. 1 illustrate the BATs with variable-width and fixed-width types of tails, respectively.

When the data is loaded from disk to main memory, MonetDB uses the exactly same data structure to represent such data on disk and in main memory. In addition, MonetDB adopts a late tuple reconstruction to save the main memory size. That is, during the entire query evaluation, all intermediate data are still the column format (i.e., the integer format instead of the actual values), and the tuples with actual values are finally reconstructed before sending the tuples to the client.

We note that the success of the column store is based on the observation that the column values belonging to the same attributes are typically the same data types and associated with similar results. Therefore, compression techniques (e.g., the adopted FOR, PFOR, PFOR-Delta [10]) can help save the storage size of the BAT (for both fixed-width and variable-width types). This immediately means that the reduced disk IO to load the compressed data. In addition, avoiding the load of all $k$ BATs belonging to a table from disk also helps optimize the disk IO.

### 2.2 Execution Model

MonetDB uses an Assembly Language (namely MAL) to program the kernel. In this way, the N-ary relational algebra plans are then translated into the BAT algebra and complied to MAL programs. After that, an *operator-at-a-time* manner is used to evaluate the MAL programs. That is, MonetDB first evaluates each operation to completion over its entire input data, and then invokes subsequent data-dependent operations. This manner allows the algebra operators to perform operational optimization. In detail, based on the entire input data, MonetDB can choose at runtime the actual algorithm and implementation to be used for better optimization. For example, a join operation can at runtime decide to perform a merge join algorithm if the input data is sorted, or a hash join otherwise; a selection operation can use a binary search if the input data is sorted, or use a hash index if available, or fall back to a scan otherwise.

### 2.3 MonetDB Architecture

The system architecture of MonetDB is shown in Fig. 2. First, MonetDB supports the standard operators such as Scan, ScanSelect, Project, Aggr, TopN, Sort, Join. During the query evaluation, a pipeline fashion is adopted by using the open(), next() and close() interfaces. The next() interface directly returns a collection of vectors (instead of a single row of tuple), where each of the vectors contains a small horizontal slice of a single column. This batch result allows the complier of MonetDB to produce data-parallel code for more efficient execution on modern CPU. Also, a *vectorized in-cache* fashion tunes the size of the vector, such that the collection of vectors needed by a query can fit the CPU cache.

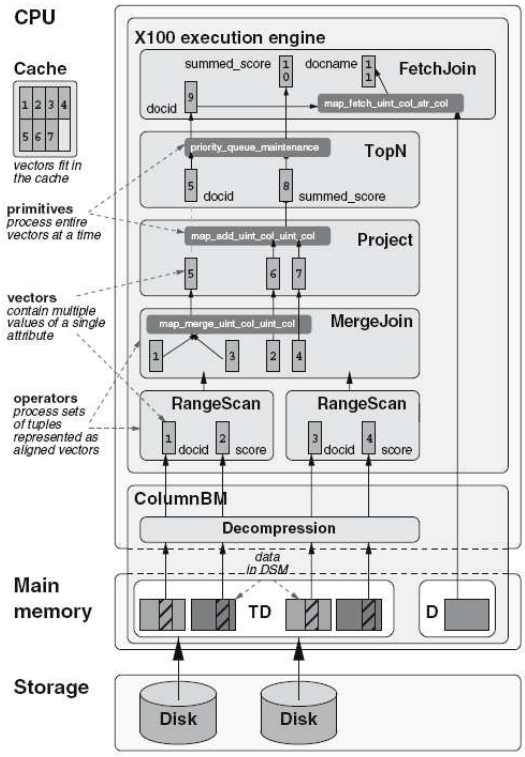


Fig. 2. MonetDB architecture [5]

Next, during the query processing, the ColumnBM buffer manager, which is based on a column-oriented storage scheme, avoids reading unnecessary columns from disk for better disk IO bandwidth utilization.

Finally, the light-weight column compression, such as FOR, PFOR and PFOR-DELTA, further improves the disk I/O-bandwidth utilization. Note that the data is decompressed on-demand, at vector granularity, directly into the CPU cache, without writing the uncompressed data back to main memory.

## 3 Applications for IR Searches

**Challenges**: Information Retrieval (IR) community has attempted to solve IR searches by database queries. However, there exist two difficulties:

- Inefficiency issue: the current RDBMS implementation cannot achieve desirable running time or suffers from poor disk resource usage.
- Inexpressiveness: the current SQL cannot offer rich abstraction for the IR search model and explicit representation of ordered data.

**Contributions**: To overcome the above difficulties, the journal paper [5] has proposed the following approaches.

- First, to help solve the inexpressiveness challenge, the paper proposed a *Matrix Framework for IR*. This framework mapped IR models to matrix spaces and designed matrix operations by treating the occurrence of terms $t \in T$ in documents $d \in D$ as a $T \times D$ matrix. The element in the matrix then is used to represent the IR score (such as TF*IDF) of such occurrence..

- Second, the paper proposed to use sparse arrays for the implementation of the Matrix Framework on relational databases, and the proposed approach is designed for efficient disk resource usage.
- Third, to efficiently answer the IR searches, the paper adopted an array query optimization technique. This technique efficiently maps the operations on sparse arrays (such as function mapping, array re-shaping, aggregation, top-N) onto relational query plan.
- Next, the proposed compression scheme also improved the query evaluation performance by optimizing the disk IO throughput.
- Finally, by the TREC TeraByte track data set, the performance of the propose solution outperforms the previous custom-built IR systems on comparable hardware.

### 3.1 Sparse relational array mapping (SRAM)

This subsection begins with a very brief introduce the storage of sparse arrays in a RDBMS and next presents the details of the SRAM mapping.

The generic approach is to store every sparse array by a relation and thus the array-cells by tuples. After that, both the standard relational indexing structures and explicit tuple clustering/sorting can then optimize the data access.

The SRAM tool is to map (i) sparse arrays to relations and (ii) array operations to relational expressions. To show the mapping, we need to know the syntax of the SRAM itself which defines the following operations over arrays: direct construction, enumeration construction, array nesting, aggregations (such as sum, prod, min, max), top-N and macro functions.

After that, array queries are transformed by the following steps: (i) from array algebra to relational algebra, and (ii) from relational algebra to a RDBMS, that are introduced as follow.

The transformation from array algebra to relational algebra supports the following operations:

1) The Apply operation is the selection of array values by their array positions. In addition, Pivot, RangeSel, and Replicate operations can be also implemented by the Apply operation. Those operations are called Shape-only array operations.
2) The Map operation between two arrays corresponds to relational join. Aggregate and top-N are also supported.
3) Simplification rules are considered to simplify the generic translation of function mapping and aggregation operations. For example, a redundant expression in the original Map transformation rule is removed for less space cost.
4) Arithmetic optimization is helpful to limit the increased complexity of generic sparse array evaluation. The optimization is to identify some common patterns such as $(x * 0) = 0, (x * 1) = x, (x/1) = x, log(1) = 0$. Such identification next activates the above Simplification rule to removes predictable computations from all translation rules.

Next, the transformation from relational algebra to a RDBMS is to express the purely relational query tree (generated by the above transformation from array algebra to relational algebra) by the query language offered by the RDBMS at hand. Note that the transformation itself is trivial and the key is the optimization of the SQL expressions transformed from relational algebra. To this end, the paper proposed two following techniques.

1) Enforcement of integrity constraints: After the two Arrays $DT$ and $S$ are stored as relations $DT(d,t,v)$ and $S(d,v)$, the integrity constraints exist on index columns: (i) $DT(d,t)$ is the primary key of relation $DT$; (ii) $S(d)$ is the primary key of relation $S$; and (iii) $DT(d)$ is a foreign key for the primary key $S(d)$. The constraints help improve cardinality estimations.
2) Creating access patterns: SRAM adopts the policy to sort the tuples of transformed relations on their primary key. This leads to creating a clustered index for each of these relations. Such an index improves efficiency dramatically (e.g., optimized for a MergeJoin algorithm).

### 3.2 A running example

We consider the following basic BM25 query:

$$\begin{array}{lcl} D20 & := & topN([s(d)|d], 20, DESC) \\ S20 & := & [s(d)|d](D20) \end{array}$$

The above query means that the indices of the 20 highest scored documents are first retrieved from the array $[s(d)|d]$ by the $TopN()$ construct and materialized as array $D20$. Then, the scores of those documents are fetched by dereferencing the array $[s(d)|d]$ with $D20$.

Based on the MonetDB transformation rules, the above query is transformed to the following physical query plan:

```
TopN(
 DenseAggr(
  Project(
   FetchJoin(
    FetchJoin(
     MergeJoin(Scan(Q), TD, TD.termid = Q.termid), T, T.termid = Q.termid),
    D, D.docid = TD.docid),
   [D.docid, scores = BM25(TD.tf, D.doclen, T.ftd)]),
  [score = sum(scores)]),
 [scoreDESC], 20)
```

### 3.3 Experiments

#### 3.3.1 Experimental Setup

- **Data Set**: TREC TeraByte track (i.e., the GOV2 collection) has 25 million web documents, with a total size of 426GB. System efficiency is measured by total execution time of 50,000 queries, and effectiveness is evaluated by early precision (p@20) on a subset of 50 preselected queries. In the experiments, the BM25 formula is used for IR model to compute the term weight (including term frequency and inverse document frequency, i.e., tf*idf).
- **Backend Database**: MonetDB/X100 is used as the experimental database due to two unique properties: i) column-oriented storage manager that provides transparent light-weight data compression (i.e., PFOR and PFOR-DELTA compression algorithm) and ii) vectorized in-cache query execution to achieve good CPU utilization.

| Run | Index size (GB) | p@20 | CPUs | Time per query (ms) |
|---|---|---|---|---|
| Indri | 100 | 0.5610 | 1 | 1724 |
| Wumpus | 14 | 0.5310 | 1 | 91 |
| Zettair | 44 | 0.4770 | 1 | 390 |
| MonetDB/X100 | 9 | 0.5470 | 1 | 117 |

Fig. 3. Performance comparison with other works [5]

#### 3.3.2 Experimental Results

Fig.3 summarizes the efficiency and precision of MonetDB and three custom IR engines. All four schemes were performed on the same hardware with the 2006 TeraByte Track as input data. This figure indicates that MonetDB results are competitive, with only slightly less efficiency than Wumpus (117 vs. 91 ms on cold data), and less precision than Indri (0.561 vs 0.547). However, Indri suffered from a significant cost in terms of execution time (117 vs 1,724 ms).

## 4 Related Work and Discussion

### 4.1 MIT Column Store [9], [1], [2]

C-Store is the first column store to comprehensively implement the columnar-oriented database system. It contains writable store (WS) and read-optimized store (RS), both of which are implemented by column store. After data is inserted to WS, the tuple mover (TM) periodically merges the updates to RS. Designed on a grid computing environment, C-Store supports transactions which include high availability and snapshot isolation for read-only transactions.

There are differences between C-Store and MonetDB. C-Store maps a table to projects, and thus allows redundant columns that appear inside multiple projects. Each column in the projects is stored with the column-wise storage layout. Second, the hybrid architecture (i.e., RS/WS/TM components) in C-Store allow the optimization of both write and read operations. Finally, high-availability is supported in C-Store.

### 4.2 Column Store in Microsoft SQL Server 2012 [8]

Microsoft SQL Server is a general-purpose database system that was designed to store data in row format. The recent 2012 version supports columnar storage (as a column store index) and efficient *batch-at-a-time* processing. As a result, SQL Server 2012 supports an index stored row-wise in a B-tree or column-wise in a column store index. Next, the batch-at-a-time processing loads a large number of rows which greatly reduce CPU time and cache misses on modern processors. Note that SQL server 2012 supports only a subset of the query operators such as scan, filter, project, hash (inner) join and (local) hash aggregation.

When compared with MonetDB which fully supports the column store, the SQL server 2012 allows only the column index and it is unclear whether the underlying storage layout of data values is also designed for the column storage.

### 4.3 Main-memory Hybrid Column Store[6]

HYRISE [6] is a main memory database system and the key idea is to automatically partition tables into vertical partitions of varying widths (depending on the access patterns of the table columns). That is, the columns involving frequent OLTP-style queries favor wider partitions (such that such columns look like the ones in traditional row-based tables. Instead, for the columns frequently accessed as a part of OLAP analytical queries, HYRISE creates narrow partitions for such columns (similar to the column storage of MonetDB).

### 4.4 Google BigTable [4]

The Google BigTable is designed to scale for petabytes of structured data and thousands of commodity servers. The data model of BigTable is a sparse, distributed, persistent multi-dimensional sorted map. The map is then indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes:
$(row : string, column : string, time : int64) \rightarrow string$.

BigTable allows client to group multiple column families together into a *locality group*. The locality group is generated as a separate SSTable (an internal file of BigTable to store data). For those column families that are not typically accessed together, BigTable creates separate locality groups for more efficient read. Moreover, the SSTables for locality groups are allowable to be compressed for less space cost and higher disk IO.

Compared with the above column store, BigTable shares some similarities. For example, locality groups in BigTable are also similarly optimized with compression for better disk read performance. Both C-Store and BigTable are designed for shared-nothing machines. By using two different data structures, one for recent writes, and one for storing long-lived data, they both move data from one form to the other. In this way, they both provide good performance on read-intensive and write-intensive applications.

In terms of the difference, column stores such as C-store and MonetDB are designed to provide database-alike APIs; while BigTable provides only low-level APIs. Moreover, the locality groups of BigTable do not support CPU- cache-level optimizations that are used in MonetDB.

### 4.5 Discussion

Table 1 summarizes the above comparison and discussion. Furthermore, we discuss the potential weakness and extension of column store as follows.

Recently, data mining and machine learning techniques are popularly used in Web searches. Though the column layout is designed for the data analysis (such as the batch scan, aggregation), it is unclear how it is used to improve the efficiency of the data mining and machine learning problems. For Web documents having a large number of document terms, such terms correspondingly indicate a larger number attributes. Though the batch-alike query processing technique can benefit the scan of a column, how such a technique can significantly improve the the classic data mining and machine problems

is unclear. For example, to cluster a set of documents, the computation of the pairwise similar of documents needs to process a large number of documents. Such a process involves the process of the inverted lists associated with a large number of terms (and thus a large number of columns). The scan obviously might still suffer from high disk IO caused by the scan of such a number of columns, although the scan inside a column still benefits from the column layout.

In addition, nowadays the modern memory capacity becomes larger and accommodates a large amount of data. It is particularly true that many in-memory databases come out to offer high performance. This partially offsets the benefits achieved by the column layout. For example, compression of data can reduce the disk IO. However, if data is inside the memory, the disk IO is not the performance bottleneck. Instead, the decompression will incur high overhead.

## 5 Conclusion

In this seminar report, we summarize the techniques of MonetDB including its column design, query processing and architecture, and the application of MonetDB in IR searches. Also, we discuss and compare MonetDB with other column databases. All these works indicate the column fashion offers a promising solution for data intensive analysis. In particular, the Google BigTable incorporated the column design, and this verified that the column design has been widely accepted in both database communities and also IR communities.

## References

[1] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column oriented database systems. *PVLDB*, 2(2):1664–1665, 2009.

[2] D. J. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD Conference*, pages 967–980, 2008.

[3] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data (awarded best paper!). In *OSDI*, pages 205–218, 2006.

[5] R. Cornacchia, S. Héman, M. Zukowski, A. P. de Vries, and P. A. Boncz. Flexible and efficient ir using array databases. *VLDB J.*, 17(1):151–168, 2008.

[6] M. Grund, P. Cudré-Mauroux, and S. Madden. A demonstration of hyrise - a main memory hybrid storage engine. *PVLDB*, 4(12):1434–1437, 2011.

[7] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[8] P.-Å. Larson, E. N. Hanson, and S. L. Price. Columnar storage in sql server 2012. *IEEE Data Eng. Bull.*, 35(1):15–20, 2012.

[9] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. 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*, pages 553–564, 2005.

[10] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, page 59, 2006.

| | MonetDB | C-Store | MSSQL'12 | HYRISE | BigTable |
|---|---|---|---|---|---|
| Column Layout | A BAT per attribute | a table maps projects, and each attribute in a project maps a column layout | Supports only column index | one table maps multiple projects | one table contains CFs and multiple CFs are as a locality group. |
| Compression Design | For, PFor, PFor-Delta | null suppression,dictionary encoding,running length encoding, bit-vector encoding, Lempel-Ziv | Dictionary compression (for String),RLE compression or bit packing | dictionary-based compression currently | a two-pass custom compression scheme: Bentley and McIlroy's scheme for long common strings across a large window, and fast compression algorithm that looks for repetitions in a small 16 KB window of the data |
| Query Processing | an operator-at-a-time for runtime optimization decision. | Compression-aware Selinger query optimization | Batch Mode Processing for query operators, such as scan, filter, project, hash (inner) join and (local) hash aggregation | joins queries:both early and late materialization (i.e., position or value-based operators). non-join queries: first position and then lookup values. Currently single-threaded and handles one operator at a time only. | not directly supported |
| Transaction Support | complete support for transactions in compliance with the SQL:2003 standard | high availability and snapshot isolation for read-only transactions | complete supports for ACID transactions | currently lacks support for transactions and recovery | single-row transactions, which can be used to perform atomic read-modify-write sequences on data stored under a single row key, and not currently support general transactions across row keys |

TABLE 1
Comparison of related works