MonetDB:
Open-source Columnar Database Technology
Beyond Textbooks

http://www.monetdb.org/

Stefan Manegold
Stefan.Manegold@cwi.nl
http://homepages.cwi.nl/~manegold/
Why?
Why?

Motivation (early 1990s)

- Relational DBMSs dominate since the late 1970's / early 1980's
  - IBM DB2, MS SQL Server, Oracle, Ingres, ...
  - Transactional workloads (OLTP, row-wise access)
  - I/O based processing

- But:
  - Workloads change (early 1990s)
  - Hardware changes (late 1990s)
  - Data “explodes” (early 2000s)
### Workload changes: Transactions (OLTP) vs ...

<table>
<thead>
<tr>
<th>contract</th>
<th>client</th>
<th>date</th>
<th>name</th>
<th>price</th>
<th>city</th>
<th>product</th>
</tr>
</thead>
<tbody>
<tr>
<td>12302346</td>
<td>10042334</td>
<td></td>
<td>Enro</td>
<td></td>
<td>Redmond</td>
<td>Car</td>
</tr>
<tr>
<td>37611373</td>
<td>10987097</td>
<td></td>
<td>Gotz</td>
<td></td>
<td>Berkeley</td>
<td>Redmond</td>
</tr>
</tbody>
</table>

- **update query:**
  - find client 10032112

- **lookup query:**
  - OLTP queries: access all columns of just one row.

<table>
<thead>
<tr>
<th>contract</th>
<th>client</th>
<th>date</th>
<th>name</th>
<th>price</th>
<th>city</th>
<th>product</th>
</tr>
</thead>
<tbody>
<tr>
<td>95371001</td>
<td>10032112</td>
<td></td>
<td>Chen</td>
<td></td>
<td>Seattle</td>
<td>House</td>
</tr>
<tr>
<td>51213123</td>
<td>10032423</td>
<td></td>
<td>Jones</td>
<td></td>
<td>Washington</td>
<td>Travel</td>
</tr>
<tr>
<td>54535545</td>
<td>10087823</td>
<td></td>
<td>Smith</td>
<td></td>
<td>New York</td>
<td>House</td>
</tr>
<tr>
<td>45447894</td>
<td>10013232</td>
<td></td>
<td>Doe</td>
<td></td>
<td>Boston</td>
<td>Car</td>
</tr>
</tbody>
</table>
Why?

Workload changes: ... vs OLAP, BI, Data Mining, ...

<table>
<thead>
<tr>
<th>contract</th>
<th>client</th>
<th>date</th>
<th>claim</th>
<th>city</th>
<th>product</th>
</tr>
</thead>
</table>

OLAP query: accesses only a few columns of almost all rows.

- select those tuples sold after march 21
- sum claims
- while grouping by city and product
Databases hit The Memory Wall


- CPU is 60%-90% idle, waiting for memory:
  - L1 data stalls
  - L1 instruction stalls
  - L2 data stalls
  - TLB stalls
  - Branch mispredictions
  - Resource stalls
Hardware Changes: The Memory Wall

Why?

Trip to memory = 1000s of instructions!
Why?

Hardware Changes: Memory Hierarchies

Latencies:
- TLB miss: 5–60 cycles
- L1 hit: 1–2 cycles
- L1 miss: 6–20 cycles
- L2 miss: 40–100 cycles
Why?

Hardware Changes: Memory Hierarchies

- Caches trade off capacity for speed
- Exploit instruction/data locality
- Demand fetch/wait for data

[ADH99]:
- Running top 4 database systems
- At most 50% CPU utilization

+Transition Lookaside Buffer (TLB) Cache for VM address translation ➔ only 64 entries!
What?
What?

MonetDB

- Database kernel developed at CWI since 1993
  - Research prototype turned into open-source product
- Pioneering columnar database architecture
  - Complete Relational/SQL (& XML/XQuery) DBMS
- Focusing on in-memory processing
  - Data is kept persistent on disk and can exceed memory limits
- Aiming at OLAP, BI, data mining & scientific workloads (“read-dominated”)
  - Supporting ACID transactions (WAL, optimistic CC)
- Platform for database architecture research
  - Used in academia (research & teaching) & commercial environments
- Back-end for various DB research projects:
  - Multi-Media DB & IR (“Tijah”), XML/XQuery (“Pathfinder”),
  - Data Mining (“Proximity”), Digital Forensics (“XIRAF”), GIS (“OSM”), ...
Column-Store
formerly know as
Decomposed Storage Model

• 1985: **DSM** (Copeland et al.; SIGMOD 1985)
• 1992: First ideas and kernel for **MonetDB** (Kersten)
• 1993: **MonetDB** is born
• 1993: KDB (first commercial DSM system (??))
• 1995: Sybase IQ
• 2002: **MonetDB** goes open-source
• 2004?: Stonebraker et al. start “C-Store” project and coin DSM as “**Column-Store**”
• 2006?: Stonebraker et al. found “Vertica”; end of “C-Store” as research project
• 2008: Zukowski, Boncz, et al. (CWI) found **VectorWise** (based on MonetDB/X100)
• 2010: INGRES (now called Actian) acquires **VectorWise**
• 2011: HP acquires Vertica
• 201?: SAP HANA, IBM BLINK -> ISAO -> BLU, Oracle Database In-Memory Microsoft SQL Server Column-store indexes (“Apollo”), ...
How?
A DECOMPOSITION STORAGE MODEL

21 Support Of Multivalued Attributes

A more comprehensive data model than normalized relations might allow multivalued attributes.

22 Support Of Entities

A more comprehensive data model than the original relational model might support the notion of entities.

23 Support Of Multiple Parent Relations

A data model with more generality than relations might allow multiple parent relations, where a single record can have more than one parent.

24 Support Of Heterogeneous Records

A data model with more generality than relations might allow heterogeneous records, where records of a single relation can have different types.

25 Support Of Directed Graphs

A data model with more generality than relations might allow a directed graph structure.

The DSM offers simplicity. Simple systems have several major advantages over complex systems. One advantage is that a set of fewer and simpler functions, given fixed development resources, can be either further tuned in software or pushed further into hardware to improve performance. This is similar to the RISC (Patterson and Ditzel 1980) approach in general purpose architectures. A second advantage is that many alternative cases with different processing strategies can less often be exploited, since the cases are not always recognized. Studies have compared the performance of transposed storage models with the NSM (Hoffer 1976, Datory 1979, March and Severance 1977, March and Scudder 1984). In this report, we describe the advantages of a fully decomposed storage model (DSM), which is a transposed storage model with surrogates included. The DSM pairs each attribute value with the surrogate of its conceptual schema record in a binary relation. For example, the above relation would be stored as:

<table>
<thead>
<tr>
<th>s1</th>
<th>sur</th>
<th>val</th>
<th>a2</th>
<th>sur</th>
<th>val</th>
<th>a3</th>
<th>sur</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>v11</td>
<td></td>
<td>s1</td>
<td>v21</td>
<td></td>
<td>s1</td>
<td>v31</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>v12</td>
<td></td>
<td>s2</td>
<td>v22</td>
<td></td>
<td>s2</td>
<td>v32</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>v13</td>
<td></td>
<td>s3</td>
<td>v23</td>
<td></td>
<td>s3</td>
<td>v33</td>
<td></td>
</tr>
</tbody>
</table>
Storing Relations in MonetDB

DSM => Column-store

Front-End

logical data model

mapping rules

physical data model (BATs)

Monet
Storing Relations in MonetDB

DSM $\Rightarrow$ Column-store

Virtual OID: seqbase=1000 (increment=1)
BAT Data Structure (old)

BAT: binary association table

BUN: binary unit

Head & Tail:
- consecutive memory blocks (arrays)
- memory-mapped files

Tail Heap:
- best-effort duplicate elimination for strings (~ dictionary encoding)
- minimal offset width; adapts automatically (1, 2, 4, 8 byte wide)

Hash tables, T-trees, R-trees, Column imprints, Ordered indexes, Database cracking
BAT Data Structure (new)

- Hash tables,
- T-trees,
- R-trees,
- Column imprints,
- Ordered indexes,
- Database cracking

Built automatically on-the-fly when required/beneficial
Kept for (later) re-use also for intermediate results

Tail:
- Consecutive memory block (array)
- Memory-mapped files

Tail Heap:
- Best-effort duplicate elimination for strings (~ dictionary encoding)
- Minimal offset width; adapts automatically (1, 2, 4, 8 byte wide)
RISC Relational Algebra

```sql
SELECT id, name, (age-30)*50 as bonus
FROM people
WHERE age > 30
```

<table>
<thead>
<tr>
<th>people_id (void)</th>
<th>people_name (str)</th>
<th>people_age (void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Alice</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Ivan</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Peggy</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Victor</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Eve</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Walter</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>Trudy</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>Bob</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>Zoe</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>Charlie</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>37</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>45</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>19</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>27</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>29</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>42</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>35</td>
<td>9</td>
</tr>
</tbody>
</table>
RISC Relational Algebra

```
SELECT id, name, (age-30)*50 as bonus
FROM people
WHERE age > 30
```
**SELECT** id, name, (age-30)*50 as bonus
**FROM** people
**WHERE** age > 30

### VIEWS
(not materialized)
SELECT id, name, (age-30)*50 as bonus 
FROM people 
WHERE age > 30

```
baticlalc_minus_int(int* res, 
   int* col, 
   int val, 
   int n) 
{
   for(i=0; i<n; i++)
       res[i] = col[i] - val;
}
```

CPU 😊? Give it “nice” code!
- few dependencies (control, data) 
- CPU gets out-of-order execution 
- compiler can e.g. generate SIMD

One loop for an entire column
- no per-tuple interpretation 
- arrays: no record navigation 
- better instruction cache locality

Simple, hard-coded semantics in operators
### SELECT id, name, (age-30)*50 as bonus FROM people WHERE age > 30

<table>
<thead>
<tr>
<th>people_id (void)</th>
<th>people_name (str)</th>
<th>people_age (void)</th>
<th>(void) (oid) (int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1 2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2 3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3 4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4 5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5 6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6 7</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7 8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8 9</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9 10</td>
</tr>
</tbody>
</table>

### RISC Relational Algebra

**Materialized Intermediate Results**

- **MarkH()**
- **MarkT()**

<table>
<thead>
<tr>
<th>sel_age (oid) (int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 37</td>
</tr>
<tr>
<td>2 45</td>
</tr>
<tr>
<td>5 31</td>
</tr>
<tr>
<td>8 42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sel_bonus (void) (int)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 350</td>
</tr>
<tr>
<td>1 750</td>
</tr>
<tr>
<td>2 50</td>
</tr>
<tr>
<td>3 600</td>
</tr>
<tr>
<td>4 250</td>
</tr>
</tbody>
</table>
SELECT id, name, (age-30)*50 as bonus
FROM people
WHERE age > 30
How is MonetDB Different

• full vertical fragmentation: always!
  • everything in binary (2-column) tables (Binary Association Table)
  • saves you from table scan hell in OLAP and Data Mining

• RISC approach to databases
  • simple back-end data model (BATs)
  • simple back-end query language (binary/columnar relational algebra: MAL)
  • no need (to pay) for a buffer manager => manage virtual memory
  • admission control in scheduler to regulate memory consumption
  • explicit transaction management => DIY approach to ACID

• Multiple user data models & query languages
  • SQL, XML/XQuery, (RDF/SPARQL)
  • front-ends map data models to BATs and query languages to MAL
• operator-at-a-time bulk processing
  • avoids tuple-at-a-time management overhead

• CPU and memory cache optimized
  • Techniques adopted from scientific programming
  • Data structures:
    • Arrays
  • Code:
    • Compiler-friendly, branch-free, loop unrolling, instruction cache friendly
  • Algorithms:
    • Exploit spatial & temporal access locality
The MonetDB Software Stack

Front-ends
- XQuery
- SQL 03
- RDF
- Arrays

Back-end(s)
- MonetDB 4
- MonetDB 5

Kernel
- MonetDB kernel
The MonetDB Software Stack

Front-ends
- XQuery
- SQL 03

Optimizers

Back-end(s)
- MonetDB 4
- MonetDB 5

Kernel
- MonetDB kernel

Strategic optimization: Rel.Alg. -> MAL
Tactical optimization: MAL -> MAL rewrites
Runtime operational optimization
RISC Relational Algebra

```
SELECT id, name, (age-30)*50 as bonus
FROM people
WHERE age > 30
```

```
_01 := sql.bind("sys","people","age",0);
_02 := algebra.select(_01,30,nil);
_03 := algebra.markH(_02,0);
_04 := algebra.markT(_02,0);
_05 := batcalc._(-_04,30);
_06 := batcalc._(*(_05,50))
_07 := sql.bind("sys","people","name",0);
_08 := algebra.join(_03,_07);
_09 := sql.bind("sys","people","id",0);
_10 := algebra.join(_03,_09);
```
PLAN SELECT a FROM t WHERE c < 10;

project (
  select (  
    table(sys.t) [ t.a, t.c, t.%TID% NOT NULL ]  
  ) [ t.c < convert(10) ]  
) [ t.a ]
EXPLAIN SELECT a FROM t WHERE c < 10;

function user.s1_1():void;
barrier _55 := language.dataflow();
  _02:bat [:void,:int] := sql.bind("sys","t","c",0);
  _07:bat [:oid,:int] := algebra.thetauselect(_02,10,"<");
  _10:bat [:oid,:void] := algebra.markT(_07,0@0);
  _11:bat [:void,:oid] := bat.reverse(_10);
  _12:bat [:oid,:int] := sql.bind("sys","t","a",0);
  _14:bat [:void,:int] := algebra.leftjoin(_11,_12);
exit _55;
  _15 := sql.resultSet(1,1,_14);
sql.rsColumn(_15,"sys.t","a","int",32,0,_14);
  _21 := io.stdout();
sql.exportResult(_21,_15);
end s1_1;
PLAN SELECT a, z FROM t, s WHERE t.c = s.x;

project (join (table(sys.t) [ t.a, t.c, t.%TID% NOT NULL ],
     table(sys.s) [ s.x, s.z, s.%TID% NOT NULL ]
  ) [ t.c = s.x ]
) [ t.a, s.z ]
EXPLAIN SELECT a, z FROM t, s WHERE t.c = s.x;

function user.s2_1():void;

barrier _73 := language.dataflow();
_02:bat[:void,:int] := sql.bind("sys","t","c",0);
_07:bat[:void,:int] := sql.bind("sys","s","x",0);
_10:bat[:int,:void] := bat.reverse(_07);
_11:bat[:oid, :oid] := algebra.join(_02, _10);
_13:bat[:oid,:void] := algebra.markT(_11,0@0);
_14:bat[:void,:oid] := bat.reverse(_13);
_15:bat[:void,:int] := sql.bind("sys","t","a",0);
_17:bat[:void,:int] := algebra.leftjoin(_14, _15);
_18:bat[:oid, :oid] := bat.reverse(_11);
_19:bat[:oid,:void] := algebra.markT(_18,0@0);
_20:bat[:void,:oid] := bat.reverse(_19);
_21:bat[:void,:int] := sql.bind("sys","s","z",0);
_23:bat[:void,:int] := algebra.leftjoin(_20, _21);
exit _73;
_24 := sql.resultSet(2,1,_17);
sql.rsColumn(_24,"sys.t","a","int",32,0,_17);
sql.rsColumn(_24,"sys.s","z","int",32,0,_23);
_33 := io.stdout();
sql.exportResult(_33,_24);
end s2_1;
Multi-core Parallelism: *Mitosis*

- Horizontally slice largest table
  - As many slices as CPU cores
  - As many slices such that #cores slices fit in memory
- Replicate query plan per slice
  - As far as possible
- Evaluate replicated plan fragments concurrently
- Concatenate partial intermediate result
- Evaluate remaining query plan
EXPLAIN SELECT a FROM t WHERE c < 10;

function user.s1_1():void;
barrier _55 := language.dataflow();
  _02a:bat[:void,:int] := sql.bind("sys","t","c",0,0,2);
  _07a:bat[:oid,:int] := algebra.thetauselect(_02a,10,"<");
  _10a:bat[:oid,:void] := algebra.markT(_07a,0@0);
  _11a:bat[:void,:oid] := bat.reverse(_10a);
  _12a:bat[:oid,:int] := sql.bind("sys","t","a",0,0,2);
  _14a:bat[:void,:int] := algebra.leftjoin(_11a,_12a);
  _02b:bat[:void,:int] := sql.bind("sys","t","c",0,1,2);
  _07b:bat[:oid,:int] := algebra.thetauselect(_02b,10,"<");
  _10b:bat[:oid,:void] := algebra.markT(_07b,0@0);
  _11b:bat[:void,:oid] := bat.reverse(_10b);
  _12b:bat[:oid,:int] := sql.bind("sys","t","a",0,1,2);
  _14b:bat[:void,:int] := algebra.leftjoin(_11b,_12b);
exit _55;
  _14 := mat.pack(_14a,_14b);
  _15 := sql.resultSet(1,1,_14);
sql.rsColumn(_15,"sys.t","a","int",32,0,_14);
  _21 := io.stdout();
sql.exportResult(_21,_15);
end s1_1;
EXPLAIN SELECT a FROM t WHERE c < 10;

function user.s1_1():void;
barrier _55 := language.dataflow();
_02a:bat[:void,:int] := sql.bind("sys","t","c",0,0,2);
_02b:bat[:void,:int] := sql.bind("sys","t","c",0,1,2);
_07a:bat[:oid, :int] := algebra.thetauselect(_02a,10,"<");
_07b:bat[:oid, :int] := algebra.thetauselect(_02b,10,"<");
_10a:bat[:oid, :void] := algebra.markT(_07a,0@0);
_10b:bat[:oid, :void] := algebra.markT(_07b,0@0);
_11a:bat[:void, :oid] := bat.reverse(_10a);
_11b:bat[:void, :oid] := bat.reverse(_10b);
_12a:bat[:oid, :int] := sql.bind("sys","t","a",0,0,2);
_12b:bat[:oid, :int] := sql.bind("sys","t","a",0,1,2);
_14a:bat[:void, :int] := algebra.leftjoin(_11a, _12a);
_14b:bat[:void, :int] := algebra.leftjoin(_11b, _12b);
exit _55;
_14 := mat.pack(_14a, _14b);
_15 := sql.resultSet(1,1,_14);
sql.rsColumn(_15,"sys.t","a","int",32,0,_14);
_21 := io.stdout();
sql.exportResult(_21,_15);
end s1_1;
Open-Source Development

- Feature releases: 3-4 per year
  - Research results
  - User requests
- Bug-fix releases: monthly
- QA
  - Automated nightly testing on >20 platforms
  - Ensure correctness & stability
  - Ensure portability
  - Bug reports become test cases
  - Semi-automatic performance monitoring
  - Passed static code verification by Coverity with only minor problems
MonetDB vs Traditional DBMS Architecture

- Architecture-Conscious Query Processing
  - vs Magnetic disk I/O conscious processing
  - Data layout, algorithms, cost models

- RISC Relational Algebra (operator-at-a-time)
  - vs Tuple-at-a-time Iterator Model
  - Faster through simplicity: no tuple expression interpreter

- Multi-Model: ODMG, SQL, XML/XQuery, ..., RDF/SPARQL
  - vs Relational with Bolt-on Subsystems
  - Columns as the building block for complex data structures

- Decoupling of Transactions from Execution/Buffering
  - vs ARIES integrated into Execution/Buffering/Indexing
  - ACID, but not ARIES.. Pay as you need transaction overhead.

- Run-Time Indexing and Query Optimization
  - vs Static DBA/Workload-driven Optimization & Indexing
  - Extensible Optimizer Framework;
  - cracking, recycling, sampling-based runtime optimization