Mammals Flourished Long Before Dinosaurs Became Extinct

VLDB 2009 Lyon - Ten Year Award
“Database Architecture Optimized For The New Bottleneck: Memory Access” (VLDB 1999)

Stefan Manegold (manegold@cwi.nl)
Peter Boncz (boncz@cwi.nl)
Martin Kersten (mk@cwi.nl)
MICHAEL STONEBRAKER

… What I see happening is that the large database vendors, whom I’ll call the elephants, are selling a one-size-fits-all, 30-year-old architecture that dates from somewhere in the late 1970s. …
MICHAEL STONEBRAKER

… What I see happening is that the large database vendors, whom I'll call the elephants, are selling a one-size-fits-all, 30-year-old architecture somewhere in the late 1970s.
Reptiles

Mammals

200 MYA
Reptiles → Dinosaur → Mamals

200 MYA
Reptiles

200 MYA

Dinosaur
Reptiles

200 MYA

Mamals

Dinosaurs
Reptiles

200 MYA

Dinosaur

60 MYA

Mammals
Reptiles  
200 MYA

Mammals

Dinosaurs

200 MYA
Reptiles

200 MYA

Mammals

60 MYA

Dinosaur
Large mammals once dined on dinosaurs

Repenomamus giganticus

Repenomamus robustus

Evolution

It is not the strongest of the species that survives, nor the most intelligent, but the one most responsive to change.

Charles Darwin (1809 - 1882)
The genes of a species

- SQL86, SQL92, SQL99, SQL03
- n-ary storage scheme
- relational algebra + DDL
- 5+ way indexing schemes
- slotted pages of records
- Volcano-style computation
The evolution of the Fox

1979-1985
Troll a relational engine to simplify relational database programming

SWI Prolog made a much better relational engine then my first system and Ingres, Oracle…
The evolution of the Fox

1979-1985
Troll a relational engine to simplify relational database programming

SWI Prolog made a much better relational engine then my first system and Ingres, Oracle…


The evolution of the Fox

1979-1985
Troll a relational engine to simplify relational database programming

Non-first-normal-form disease
Object-orientation religion
IO pages size increase
The evolution of the Fox

1979-1985
Troll a relational engine to simplify relational database programming

Non-first-normal-form disease
Object-orientation religion
IO pages size increase

512 bytes
The evolution of the Fox

1979-1985
Troll a relational engine to simplify relational database programming

Non-first-normal-form disease
Object-orientation religion
IO pages size increase
The evolution of the Fox

1979-1985
Troll a relational engine to simplify relational database programming

Non-first-normal-form disease
Object-orientation religion
IO pages size increase
Albert Einstein

“We can't solve problems by using the same kind of thinking we used when we created them.”
A DECOMPOSITION STORAGE MODEL

SIGMOD 1985

George P. Copeland
Setrag N. Khoshafian

Microelectronics And Technology Computer Corporation
9430 Research Blvd
Austin, Texas 78759

Abstract

This report examines the relative advantages of a storage model based on decomposition (of community view relations into binary relations containing a surrogate and one attribute) over conventional n-ary storage models.

There seems to be a general consensus among the database community that the n-ary approach is better. This conclusion is usually based on a consideration of only one or two dimensions of a database system. The purpose of this report is not to claim that decomposition is better. Instead, we claim that the consensus opinion is not well founded and that neither is clearly better until a closer analysis is made along the many dimensions of a database system. The purpose of this report is to move further in both scope and depth toward such an analysis. We examine such dimensions as simplicity, generality, storage requirements, update performance and retrieval performance.

Some database systems use a fully transposed storage model, for example, RM (Lorie and Symonds 1971), TOD (Wiederhold et al 1975), RAPID (Turner et al 1979), ALDS (Burnett and Thomas 1981), Delta (Shibayama et al 1982) and (Tanaka 1983). This approach stores all values of the same attribute of a conceptual schema relation together. Several studies have compared the performance of transposed storage models with the NSM (Pomerantsev 1973, Barak 1975, 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>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>s1</td>
<td>v11</td>
</tr>
<tr>
<td>s2</td>
<td>s1</td>
<td>v21</td>
</tr>
<tr>
<td>s3</td>
<td>s1</td>
<td>v31</td>
</tr>
<tr>
<td>s2</td>
<td>s2</td>
<td>v12</td>
</tr>
<tr>
<td>s2</td>
<td>s2</td>
<td>v22</td>
</tr>
<tr>
<td>s2</td>
<td>s2</td>
<td>v32</td>
</tr>
<tr>
<td>s3</td>
<td>s3</td>
<td>v13</td>
</tr>
<tr>
<td>s3</td>
<td>s3</td>
<td>v23</td>
</tr>
<tr>
<td>s3</td>
<td>s3</td>
<td>v33</td>
</tr>
</tbody>
</table>
A DECOMPOSITION STORAGE MODEL

SIGMOD 1985

2.1 Support Of Multivalued Attributes

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

2.2 Support Of Entities

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

2.3 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...

2.4 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...

2.5 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 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 (Roffer 1976, Batory 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:

| s1 | sur | val
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>v11</td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>v12</td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>v13</td>
<td></td>
</tr>
</tbody>
</table>

| s2 | sur | val
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>v21</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>v22</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>v23</td>
<td></td>
</tr>
</tbody>
</table>

| s3 | sur | val
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>v31</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>v32</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>v33</td>
<td></td>
</tr>
</tbody>
</table>
The genes of a new species

- SQL86, SQL92, SQL99, SQL03
- n-ary storage scheme
- relational algebra + DDL
- 5+ way indexing schemes
- slotted pages of records
- Volcano-style computation
The genes of a new species

- SQL86, SQL92, SQL99, SQL03

<table>
<thead>
<tr>
<th>Binary Association Tables</th>
<th>storage scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>relational algebra + DDL</td>
</tr>
<tr>
<td>Self managing Arrays</td>
<td>indexing schemes</td>
</tr>
<tr>
<td></td>
<td>of records</td>
</tr>
<tr>
<td>Materialize operator</td>
<td>computation</td>
</tr>
</tbody>
</table>
The MonetDB Software Stack

- SQL 03
- Optimizers
- MonetDB 5
- MonetDB kernel
Cache-Conscious Query Processing in

Stefan Manegold (manegold@cwi.nl)
Peter Boncz (boncz@cwi.nl)
Martin Kersten (mk@cwi.nl)
Evolution == Progress?

Simple Scan: `select max(c) from t`
Evolution == Progress?

Hardware Evolution (Moore's Law)

BUT Software Stagnation!
Databases hit The Memory Wall

- Detailed and exhaustive analysis for different workloads using 4 RDBMSs by Anastassia Ailamaki et al. in “DBMSs On A Modern Processor: Where Does Time Go?” (VLDB 1999)

- CPU is 50%-90% idle, waiting for memory:
  - L1 data stalls
  - L1 instruction stalls
  - L2 data stalls
  - TLB stalls
  - Branch mispredictions
  - Resource stalls
CPU & Hierarchical Memory System (1999)

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

2009:
- L2 (+L3) on CPU die
- Memory access: up to 1000 cycles
### Required DBMS Evolution

- Memory access has become a significant cost factor
- Database algorithms suffer particularly from latency (due to random access patterns)

<table>
<thead>
<tr>
<th>Goal</th>
<th>Optimize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use cache lines fully</td>
<td>⇒ Data structures</td>
</tr>
<tr>
<td>Prevent cache &amp; TLB misses</td>
<td>⇒ Memory access / algorithms</td>
</tr>
<tr>
<td>Prevent CPU stalls</td>
<td>⇒ Implementation techniques</td>
</tr>
<tr>
<td>Exploit CPU-inherent parallelism</td>
<td>⇒ Implementation techniques</td>
</tr>
</tbody>
</table>
## Data Structure Evolution

<table>
<thead>
<tr>
<th>A_1</th>
<th>A_2</th>
<th>A_3</th>
<th>...</th>
<th>A_n</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Row-storage wastes bandwidth" /></td>
<td><img src="image2.png" alt="Column-storage exploits full bandwidth" /></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Row-storage** wastes bandwidth
- **Column-storage** exploits full bandwidth

**requested attribute**

**cache line**
Algorithm Evolution: Joins

- Nested-loop:
  + sequential access to both inner & outer input
  -- quadratic complexity

- Sort-merge:
  + single sequential scan during merge (“benefit”)
  -- random access during sort (“investment”)

- Hash-join:
  + sequential scan over both inputs
  -- random access to hash table (build & probe)
Algorithm Evolution: Partitioned Hash-Joins

Phase 1:
- Cluster both input relations
- Create clusters that fit in CPU cache
- Restrict random data access to (smallest) cache
- Avoid cache capacity misses

Phase 2:
- Join matching clusters

non–clustered  clustered
Partitioned Hash-Join: Joining (Phase 2)

Phase 2 solved; but what about Phase 1?

Points measured
Lines modeled
[VLDB2002]
Problem:

$\$ Number of clusters exceeds number of cache lines / TLB entries
$\$ => cache / TLB thrashing

Solution:

$\$ Multi-pass clustering
Algorithm Evolution: Multi-Pass Clustering

- Limit number of clusters per pass
- Avoid cache / TLB thrashing
- Trade memory cost for CPU cost
Partitioned Hash-Join: Multi-Pass Clustering

elapsed time [seconds]

passes:
4
3
2
1

Number of clusters
Partitioned Hash-Join

![Graph showing performance of Partitioned Hash-Join with different cluster sizes and passes. The graph indicates that the performance improves as the cluster size increases and the number of passes decreases.]
Joins in Column-Stores: Handling Payload

Problem:
- Join result: pairs of tuple IDs; *Out-of order*
- => random access during projection / tuple-reconstruction

Solutions:
- Jive-Join (Li, Ross; VLDB-Journal 1998)
- Flash-Join (Tsiragiannis, Harizopoulos, Shah, Wiener, Graefe; SIGMOD 2009)
- Radix-Decluster (Boncz, Manegold, Kersten; VLDB 2004)
- (Sideways Cracking (Idreos, Kersten, Manegold; SIGMOD 2009))

=> post-projection / late materialization
Algorithm Evolution: Multi-pass Clustering

1 pass

P passes

P passes, CPU optimized
Algorithm Evolution: Partitioned Hash-Join

CPU optimized
Cost Model Evolution: Data Access

- Total data access cost is sum over all cache/memory levels
- Cost per level is number of cache misses scored by latency
- Simple tool to measure latency per cache level (“The Calibrator”)
- Few simple basic access patterns “sequential”, “random”, ...
- Compound access patterns: combinations of basic access patterns
- Basic cost functions: estimate number of cache misses of basic access patterns
- Rules how to create compound cost functions using basic cost functions
- Describe data access of algorithms using access patterns
The Bigger Picture: Evolving Columnar Database Architecture

Stefan Manegold (manegold@cwi.nl)
Peter Boncz (boncz@cwi.nl)
Martin Kersten (mk@cwi.nl)
SELECT id, name, (age-30)*50 as bonus
FROM people
WHERE age > 30

RISC Relational Algebra

Simple, hard-coded semantics in operators
RISC Relational Algebra

CPU \( \bigcup \) ? 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

```c
{  
  for(i=0; i<n; i++)
    res[i] = col[i] - val;
}
```

Simple, hard-coded semantics in operators
RISC Relational Algebra

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

MATERIALIZED intermediate results
Materialization vs Pipelining

SELECT id, name (age-30)*50 AS bonus
FROM employee
WHERE age > 30

```
next()
```

```
102  ivan  350
next()
```

```
102  ivan  37  7  350
```

```
37 > 30 ?
```

```
101  alice  22  TRUE
next()
```

```
7 * 50
```

```
102  ivan  37
```

```
SCAN
```

```
next()
```

```
MATERIALIZED intermediate results
```

```
MONETDB
```

```
Materialization vs Pipelining
```

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

```
350
FALSE
TRUE
22 > 30 ?
37 > 30 ?
37 – 30
7 * 50
7
```
MonetDB spin-off: vectorwise

Materialization vs Pipelining
```sql
SELECT
next()
PROJECT
next()
SELECT
next()
SCAN
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>alice</td>
<td></td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>ivan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>peggy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>victor</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>37</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>45</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>
"Vectorized In Cache Processing"

vector = array of ~100

CPU cache Resident

```
next()
101 102 104 105
alice ivan peggy victor
22 37 45 25
next() > 30 ?
```

```
next()
101 102 104 105
alice ivan peggy victor
22 37 45 25
next() > 30 ?
```
Observations:

next() called much less often.
more time spent in primitives
less in overhead

primitive calls process an array of values in a loop:
Observations:

next() called much less often
more time spent in primitives
less in overhead

primitive calls process an array of values in a loop:

CPU ? Give it “nice” code!

- few dependencies (control, data)
- CPU gets out-of-order execution
- compiler can e.g. generate SIMD
Hey!! stop reading this, please!

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

<table>
<thead>
<tr>
<th>&gt; 30 ?</th>
<th>FALSE</th>
<th>TRUE</th>
<th>TRUE</th>
<th>FALSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>for(i=0; i&lt;n; i++)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>res[i] = (col[i] &gt; x)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| - 30 |
| 7 15 |
| for(i=0; i<n; i++) |
| res[i] = (col[i] - x) |

| * 50 |
| 350 750 |
| for(i=0; i<n; i++) |
| res[i] = (col[i] * x) |
Observations:

next() called much less often
more time spent in primitives
less in overhead

primitive calls process an array of values in a loop:

```
for(i=0; i<n; i++)
    res[i] = (col[i] > x)
```

```
for(i=0; i<n; i++)
    res[i] = (col[i] - x)
```

```
for(i=0; i<n; i++)
    res[i] = (col[i] * x)
```

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

Loops with…

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

> 30 ?
| FALSE | TRUE | TRUE | FALSE |
- 30
| 7    | 15   |
+ 50
| 350  | 750  |

Hey!! stop reading this, please!
Loops with…

"Vectorized In Cache Processing"

vector = array of ~100 CPU cache Resident

for(i=0; i<n; i++)
res[i] = (col[i] > x)

for(i=0; i<n; i++)
res[i] = (col[i] - x)

for(i=0; i<n; i++)
res[i] = (col[i] * x)

> 30 ?
- 30
* 50

Give it "nice" code!

- few dependencies (control, data)
- CPU gets out-of-order execution
- compiler can e.g. generate SIMD

Hey!! stop reading this, please!

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

more time spent in primitives
less in overhead
called much less often

Observations:

Loops with…

"Vectorized In Cache Processing"

vector = array of ~100 CPU cache Resident

for(i=0; i<n; i++)
res[i] = (col[i] > x)

for(i=0; i<n; i++)
res[i] = (col[i] - x)

for(i=0; i<n; i++)
res[i] = (col[i] * x)
The optimal diet?

TPCH Q1

Vector Size

Time (seconds)

"tuple at a time"
DBMS "X"
MySQL 4.1
interpretation dominates execution

"column at a time"
MonetDB/MIL
main-memory materialization overhead
query without selection

"vector at a time"
MonetDB/X100
low interpretation overhead
in-cache materialization

Hand-Coded C Program

2.4
3.7

206
256
64K
256K
1M
4M
6M

0.1
0.22
0.60
1.0
10
100

CWI vectorwise
The optimal diet?

Less and less iterator.next() and primitive function calls ("interpretation overhead")
Vectors start to exceed the CPU cache, causing additional memory traffic.
Very fast

1 core consumes 5GBs/sec!

Research on achieving I/O balance

- Columnar data storage [Boncz et al., CIDR’05]
- New compression techniques [Heman et al., ICDE’06]
- Cooperative Scans [Zukowski et al., VLDB’07]
MonetDB Highlights

- **Architecture-Conscious Query Processing**
  - Data layout, algorithms, cost models
- **Multi-Model: ODMG, SQL, XQuery, .. SPARQL**
  - Columns as the building block for complex data structures
- **RISC Relational Algebra (vs CISC)**
  - Faster through simplicity: no tuple expression interpreter
- **Decoupling of Transactions from Execution/Buffering**
  - ACID, but not ARIES. Pay as you need transaction overhead.
  - differential, lazy, optimistic, snapshot
- **Run-Time Indexing and Query Optimization**
  - Extensible Optimizer Framework
  - cracking, recycling, sampling-based runtime optimization
MonetDB  vs Traditional Architecture

§ Architecture-Conscious Query Processing
  § vs Magnetic disk I/O conscious processing

§ Multi-Model: ODMG, SQL, XQuery, .. SPARQL
  § vs Relational with Bolt-on Subsystems

§ RISC Relational Algebra
  § vs Tuple-at-a-time Iterator Model

§ Decoupling of Transactions from Execution/Buffering
  § vs ARIES integrated into Execution/Buffering/Indexing

§ Run-Time Indexing and Query Optimization
  § vs Static DBA/Workload-driven Optimization & Indexing
The MonetDB Software Stack

- SQL 03: Orthogonal extension of SQL03
- Optimizers: Clear computational semantics
- MonetDB 5: Minimal extension to MonetDB
- MonetDB kernel
The MonetDB Software Stack

- XQuery
- SQL 03
- RDF
- Arrays
- Optimizers
- SOAP
- MonetDB 4
- MonetDB 5
- OGIS
- X100
- MonetDB kernel
- Compile
The MonetDB Software Stack

Extensible query language frontend framework.. SGL?

- SOAP
- MonetDB 4
- MonetDB 5
- MonetDB kernel
- SQL 03
- RDF
- Arrays
- Optimizers

vectorwise
Extensible Dynamic Runtime QOPT Framework!
Farming new species
Cyclotron
Romulo Gonçalves

Data cell
Erietta Liarou

Sky server
Milena Ivanova

Armada
Fabian Groffen

Cracking
Stratos Idreos

XRPC
Jenny Zhang

XML pattern search
Nan Tang

RDF Graphs
Lefteris Sidirourgos
Acknowledgements

Martin Kersten
Peter Boncz
Niels Nes
Stefan Manegold
Fabian Groffen
Sjoerd Mullender
Steffen Goeldner
Arjen de Vries
Menzo Windhouwer
Tim Ruhl
Romulo Goncalves

Alex van Ballegooij
Johan List
Georgina Ramirez
Marcin Zukowski
Roberto Cornacchia
Sandor Heman
Torsten Grust
Jens Teubner
Maurice van Keulen
Jan Flokstra
Milena Ivanova
Lefteris Sidirourgos

Jan Rittinger
Wouter Alink
Jennie Zhang
Stratos Idreos
Erietta Liarou
Lefteris Sidirourgos
Florian Waas
Albrecht Schmidt
Jonas Karlsson
Martin van Dinther
Peter Bosch
Carel van den Berg
Wilco Quak
Whoa MonetDB! Speed lines!