Column-Oriented Database Systems

Part 1: Stavros Harizopoulos (HP Labs)
Part 2: Daniel Abadi (Yale)
Part 3: Peter Boncz (CWI)
What is a column-store?

**row-store**

**column-store**

+ easy to add/modify a record
- might read in unnecessary data

+ only need to read in relevant data
- tuple writes require multiple accesses

=> suitable for read-mostly, read-intensive, large data repositories
Are these two fundamentally different?

1. The only fundamental difference is the storage layout
2. However: we need to look at the big picture

- different storage layouts proposed

- row-stores
- row-stores++
- row-stores++

- ‘70s ‘80s ‘90s ‘00s today

- new applications
- new bottleneck in hardware

- column-stores

How did we get here, and where we are heading

- Part 1

What are the column-specific optimizations?

- Part 2

How do we improve CPU efficiency when operating on Cs?

- Part 3
Outline

1 Part 1: Basic concepts — Stavros
   1 Introduction to key features
   1 From DSM to column-stores and performance tradeoffs
   1 Column-store architecture overview
   1 Will rows and columns ever converge?

1 Part 2: Column-oriented execution — Daniel

1 Part 3: MonetDB/X100 and CPU efficiency — Peter
Telco Data Warehousing example

1. Typical DW installation

2. Real-world example

“One Size Fits All? - Part 2: Benchmarking Results” Stonebraker et al. CIDR 2007

**QUERY 2**

```sql
SELECT account.account_number,
sum (usage.toll_airtime),
sum (usage.toll_price)
FROM usage, toll, source, account
WHERE usage.toll_id = toll.toll_id
AND usage.source_id = source.source_id
AND usage.account_id = account.account_id
AND toll.type_ind in ('AE', 'AA')
AND usage.toll_price > 0
AND source.type != 'CIBER'
AND toll.rating_method = 'IS'
AND usage.invoice_date = 20051013
GROUP BY account.account_number
```

<table>
<thead>
<tr>
<th>Query</th>
<th>Column-store</th>
<th>Row-store</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.06</td>
<td>300</td>
</tr>
<tr>
<td>2</td>
<td>2.20</td>
<td>300</td>
</tr>
<tr>
<td>3</td>
<td>0.09</td>
<td>300</td>
</tr>
<tr>
<td>4</td>
<td>5.24</td>
<td>300</td>
</tr>
<tr>
<td>5</td>
<td>2.88</td>
<td>300</td>
</tr>
</tbody>
</table>

Why? Three main factors (next slides)
Telco example explained (1/3):

**read efficiency**

**row store**

read pages containing entire rows

one row = 212 columns!

is this typical? (it depends)

**column store**

read only columns needed

in this example: 7 columns

**What about vertical partitioning?**

(it does not work with ad-hoc queries)

**caveats:**

- “select * ” not any faster
- clever disk prefetching
- clever tuple reconstruction
Telco example explained (2/3): compression efficiency

1. Columns compress better than rows
   1. Typical row-store compression ratio 1 : 3
   1. Column-store 1 : 10

1. Why?
   1. Rows contain values from different domains
      => more entropy, difficult to dense-pack
   1. Columns exhibit significantly less entropy

1. Examples: Male, Female, Female, Female, Male

1. Caveat: CPU cost (use lightweight compression)
Telco example explained (3/3): sorting & indexing efficiency

1. Compression and dense-packing free up space
   1. Use multiple overlapping column collections
   1. Sorted columns compress better
   1. Range queries are faster
   1. Use sparse clustered indexes

What about heavily-indexed row-stores? (works well for single column access, cross-column joins become increasingly expensive)
Additional opportunities for column-stores

1. Block-tuple / vectorized processing
   1. Easier to build block-tuple operators
      1. Amortizes function-call cost, improves CPU cache performance
   1. Easier to apply vectorized primitives
      1. Software-based: bitwise operations
      1. Hardware-based: SIMD

1. Opportunities with compressed columns
   1. Avoid decompression: operate directly on compressed
   1. Delay decompression (and tuple reconstruction)
      1. Also known as: late materialization

1. Exploit columnar storage in other DBMS components
   1. Physical design (both static and dynamic)

See: Database Cracking, from CWI
Effect on C-Store performance

Average for SSBM queries on C-store

- Column-oriented join algorithm
- Enable compression & operate on compressed
- Enable late materialization

“Column-Stores vs Row-Stores: How Different are They Really?” Abadi, Hachem, and Madden. SIGMOD 2008.
Summary of column-store key features

1. Storage layout
   - columnar storage
   - header/ID elimination
   - compression
   - multiple sort orders

2. Execution engine
   - column operators
   - avoid decompression
   - late materialization
   - vectorized operations

3. Design tools, optimizer
Outline

1 Part 1: Basic concepts — Stavros
   1 Introduction to key features
   1 From DSM to column-stores and performance tradeoffs
   1 Column-store architecture overview
   1 Will rows and columns ever converge?

1 Part 2: Column-oriented execution — Daniel

1 Part 3: MonetDB/X100 and CPU efficiency — Peter
From DSM to Column-stores

70s -1985:  
- DSM paper  
- "A decomposition storage model"  

1985: DSM paper  
- "A decomposition storage model"

1990s: Commercialization through SybaseIQ

Late 90s – 2000s: Focus on main-memory performance

1. DSM “on steroids” [1997 – now]  
   CWI: MonetDB

   Wisconsin: PAX, Fractured Mirrors
   - Michigan: Data Morphing
   - CMU: Clotho

2005 - : Re-birth of read-optimized DSM as “column-store”

- MIT: C-Store
- CWI: MonetDB/X100
- 10+ startups
The original DSM paper

- Proposed as an alternative to NSM
- 2 indexes: clustered on ID, non-clustered on value
- Speeds up queries projecting few columns
- Requires more storage

Memory wall and PAX

90s: Cache-conscious research

Shatdal, Kant, Naughton. VLDB 1994.

to: “Database Architecture Optimized for the New Bottleneck: Memory Access.”
Boncz, Manegold, Kersten. VLDB 1999.


PAX: Partition Attributes Across

1 Retains NSM I/O pattern
1 Optimizes cache-to-RAM communication

“Weaving Relations for Cache Performance.”
More hybrid NSM/DSM schemes

1. Dynamic PAX: Data Morphing


1. Clotho: custom layout using scatter-gather I/O

   “Clotho: Decoupling Memory Page Layout from Storage Organization.”
   Shao, Schindler, Schlosser, Ailamaki, and Ganger. VLDB 2004.

1. Fractured mirrors

   1. Smart mirroring with both NSM/DSM copies

MonetDB (more in Part 3)

Late 1990s, CWI: Boncz, Manegold, and Kersten

Motivation:
- Main-memory
- Improve computational efficiency by avoiding expression interpreter
- DSM with virtual IDs natural choice
- Developed new query execution algebra

Initial contributions:
- Pointed out memory-wall in DBMSs
- Cache-conscious projections and joins
- …
2005: the (re)birth of column-stores

1. New hardware and application realities
   1. Faster CPUs, larger memories, disk bandwidth limit
   1. Multi-terabyte Data Warehouses

1. New approach: combine several techniques
   1. Read-optimized, fast multi-column access, disk/CPU efficiency, light-weight compression

1. C-store paper:
   1. First comprehensive design description of a column-store

1. MonetDB/X100
   1. “proper” disk-based column store

1. Explosion of new products
Performance tradeoffs: columns vs. rows

DSM traditionally was not favored by technology trends
How has this changed?

1. Optimized DSM in “Fractured Mirrors,” 2002
   “Apples-to-apples” comparison
   “Performance Tradeoffs in Read-Optimized Databases”
   Harizopoulos, Liang, Abadi, Madden, VLDB’06

1. Follow-up study
   “Read-Optimized Databases, In-Depth” Holloway, DeWitt, VLDB’08

1. Main-memory DSM vs. NSM
   “DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing”
   Boncz, Zukowski, Nes, DaMoN’08

1. Flash-disks: a come-back for PAX?
   “Fast Scans and Joins Using Flash Drives” Shah, Harizopoulos, Wiener, Graefe. DaMoN’08
   “Query Processing Techniques for Solid State Drives”
   Tsirogiannis, Harizopoulos, Shah, Wiener, Graefe, SIGMOD’09
Fractured mirrors: a closer look

1. Store DSM relations inside a B-tree
   1. Leaf nodes contain values
   1. Eliminate IDs, amortize header overhead
   1. Custom implementation on Shore

```
<table>
<thead>
<tr>
<th>Tuple Header</th>
<th>TID</th>
<th>Column Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>a1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>a2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>a3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>a4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>a5</td>
</tr>
</tbody>
</table>
```


Fractured mirrors: performance

From PAX paper:

1. Chunk-based tuple merging
   1. Read in segments of M pages
   1. Merge segments in memory
   1. Becomes CPU-bound after 5 pages
Column-scanner implementation

SELECT name, age
WHERE age > 40

apply predicate(s)

row scanner

Direct I/O

prefetch ~100ms worth of data

column scanner

apply predicate #1

Performance Tradeoffs in Read-Optimized Databases
Harizopoulos, Liang, Abadi, Madden, VLDB’06
Scan performance

1. Large prefetch hides disk seeks in columns
2. Column-CPU efficiency with lower selectivity
3. Row-CPU suffers from memory stalls
4. Memory stalls disappear in narrow tuples
5. Compression: similar to narrow

not shown, details in the paper
Even more results

- Same engine as before
- Additional findings

Non-selective queries, narrow tuples, favor well-compressed rows
Materialized views are a win
Scan times determine early materialized joins

"Read-Optimized Databases, In-Depth" Holloway, DeWitt, VLDB’08

CPU-bound!

wide attributes: same as before

Columns Returned

Time (s)

- C-25%
- C-10%
- C-0.1%
- R-50%

Column-joins are covered in part 2!
Speedup of columns over rows

- Rows favored by narrow tuples and low \( cpdb \)
- Disk-bound workloads have higher \( cpdb \)

“Performance Tradeoffs in Read-Optimized Databases”
Harizopoulos, Liang, Abadi, Madden, VLDB’06
Varying prefetch size

No prefetching hurts columns in single scans
Varying prefetch size

1. No prefetching hurts columns in single scans
2. Under competing traffic, columns outperform rows for any prefetch size
**CPU Performance**

"DSM vs. NSM: CPU performance trade-offs in block-oriented query processing"
Boncz, Zukowski, Nes, DaMoN’08

1. Benefit in on-the-fly conversion between NSM and DSM
2. DSM: sequential access (block fits in L2), random in L1
3. NSM: random access, SIMD for grouped Aggregation

---

**Figure 5:** TPC-H Q1, with a varying number of keys and different data organizations (ht – hash table)
New storage technology: Flash SSDs

1. Performance characteristics
   - very fast random reads, slow random writes
   - fast sequential reads and writes

2. Price per bit (capacity follows)
   - cheaper than RAM, order of magnitude more expensive than Disk

3. Flash Translation Layer introduces unpredictability
   - avoid random writes!

4. Form factors not ideal yet
   - SSD (small reads still suffer from SATA overhead/OS limitations)
   - PCI card (high price, limited expandability)

5. Boost Sequential I/O in a simple package
   - Flash RAID: very tight bandwidth/cm³ packing (4GB/sec inside the box)

6. Column Store Updates
   - useful for delta structures and logs

7. Random I/O on flash fixes unclustered index access
   - still suboptimal if I/O block size > record size
   - therefore column stores profit much less than horizontal stores

8. Random I/O useful to exploit secondary, tertiary table orderings
   - the larger the data, the deeper clustering one can exploit
Even faster column scans on flash SSDs

1. New-generation SSDs
   1. Very fast random reads, slower random writes
   1. Fast sequential RW, comparable to HDD arrays
   1. No expensive seeks across columns

1. FlashScan and Flashjoin: PAX on SSDs, inside Postgres

30K Read IOps, 3K Write Iops
250MB/s Read BW, 200MB/s Write

“Query Processing Techniques for Solid State Drives” Tsirogiannis, Harizopoulos, Shah, Wiener, Graefe, SIGMOD’09

mini-pages with no qualified attributes are not accessed
Column-scan performance over time

- regular DSM (2001)
  - from 7x slower
- optimized DSM (2002)
  - and 3x faster!
- column-store (2006)
  - to 2x slower
- SSD Postgres/PAX (2009)
  - ..to same
- ..to 1.2x slower
Outline

1 Part 1: Basic concepts — Stavros
   1 Introduction to key features
   1 From DSM to column-stores and performance tradeoffs
   1 Column-store architecture overview
   1 Will rows and columns ever converge?

1 Part 2: Column-oriented execution — Daniel

1 Part 3: MonetDB/X100 and CPU efficiency — Peter
Architecture of a column-store

storage layout
1. read-optimized: dense-packed, compressed
2. organize in extends, batch updates
3. multiple sort orders
4. sparse indexes

engine
1. block-tuple operators
2. new access methods
3. optimized relational operators

system-level
1. system-wide column support
2. loading / updates
3. scaling through multiple nodes
4. transactions / redundancy
C-Store

1. Compress columns
2. No alignment
3. Big disk blocks
4. Only materialized views (perhaps many)
5. Focus on Sorting not indexing
6. Data ordered on anything, not just time
7. Automatic physical DBMS design
8. Optimize for grid computing
9. Innovative redundancy
10. Xacts – but no need for Mohan
11. Column optimizer and executor

C-Store: only materialized views (MVs)

1. **Projection** (MV) is some number of columns from a fact table
2. Plus columns in a dimension table – with a 1-n join between Fact and Dimension table
3. Stored in order of a storage key(s)
4. Several may be stored!
5. With a **permutation**, if necessary, to map between them
6. Table (as the user specified it and sees it) is not stored!
7. No secondary indexes (they are a one column sorted MV plus a permutation, if you really want one)

<table>
<thead>
<tr>
<th>User view:</th>
<th>Possible set of MVs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP (name, age, salary, dept)</td>
<td>MV-1 (name, dept, floor) in floor order</td>
</tr>
<tr>
<td>Dept (dname, floor)</td>
<td>MV-2 (salary, age) in age order</td>
</tr>
<tr>
<td></td>
<td>MV-3 (dname, salary, name) in salary order</td>
</tr>
</tbody>
</table>
Continuous Load and Query (Vertica)

Hybrid Storage Architecture

- **Write Optimized Store (WOS)**
  - Memory-based
  - Unsorted / Uncompressed
  - Segmented
  - Low latency / Small quick inserts

- **Read Optimized Store (ROS)**
  - On disk
  - Sorted / Compressed
  - Segmented
  - Large data loaded directly

TUPLE MOVER
Asynchronous Data Transfer

Trickle Load
Loading Data (Vertica)

- INSERT, UPDATE, DELETE
- Bulk and Trickle Loads
  - COPY
  - COPY DIRECT
- User loads data into logical Tables
- Vertica loads atomically into storage

Write-Optimized Store (WOS)  
In-memory

Read-Optimized Store (ROS)  
On-disk

Automatic Tuple Mover
Applications for column-stores

1. Data Warehousing
   1. High end (clustering)
   1. Mid end/Mass Market
   1. Personal Analytics

1. Data Mining
   1. E.g. Proximity
   1. Google BigTable
   1. RDF
     1. Semantic web data management
   1. Information retrieval
     1. Terabyte TREC

1. Scientific datasets
   1. SciDB initiative
   1. SLOAN Digital Sky Survey on MonetDB
List of column-store systems

1. Cantor (history)
2. Sybase IQ
3. SenSage (former Addamark Technologies)
4. Kdb
5. 1010data
6. MonetDB
7. C-Store/Vertica
8. X100/VectorWise
9. KickFire
10. SAP Business Accelerator
11. Infobright
12. ParAccel
13. Exasol
Outline

1. Part 1: Basic concepts — Stavros
   1. Introduction to key features
   1. From DSM to column-stores and performance tradeoffs
   1. Column-store architecture overview
   1. Will rows and columns ever converge?

1. Part 2: Column-oriented execution — Daniel

1. Part 3: MonetDB/X100 and CPU efficiency — Peter
Simulate a Column-Store inside a Row-Store

Option A: Vertical Partitioning

Option B: Index Every Column

Date Index

Store Index

Date  Store  Product  Customer  Price
01/01  BOS   Table   Mesa     $20
01/01  NYC   Chair   Lutz    $13
01/01  BOS   Bed     Mudd    $79
Simulate a Column-Store inside a Row-Store

Option A: Vertical Partitioning

```
<table>
<thead>
<tr>
<th>Date</th>
<th>Store</th>
<th>Product</th>
<th>Customer</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/01</td>
<td>BOS</td>
<td>Table</td>
<td>Mesa</td>
<td>$20</td>
</tr>
<tr>
<td>01/01</td>
<td>NYC</td>
<td>Chair</td>
<td>Lutz</td>
<td>$13</td>
</tr>
<tr>
<td>01/01</td>
<td>BOS</td>
<td>Bed</td>
<td>Mudd</td>
<td>$79</td>
</tr>
</tbody>
</table>
```

Can explicitly run-length encode date

“Teaching an Old Elephant New Tricks.”
Bruno, CIDR 2009.

Option B: Index Every Column

Date Index

Store Index

...
Experiments

1. Star Schema Benchmark (SSBM)

- Implemented by professional DBA
- Original row-store plus 2 column-store simulations on same row-store product

Adjoined Dimension Column Index (ADC Index) to Improve Star Schema Query Performance”. O’Neil et. al. ICDE 2008.

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>Normal Row-Store</th>
<th>Vertically Partitioned Row-Store</th>
<th>Row-Store With All Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>25.7</td>
<td>79.9</td>
<td>221.2</td>
</tr>
</tbody>
</table>
What’s Going On? Vertical Partitions

1. Vertical partitions in row-stores:
   1. Work well when workload is known
   1. ...and queries access disjoint sets of columns
   1. See automated physical design

2. Do not work well as full-columns
   1. TupleID overhead significant
   1. Excessive joins

“Column-Stores vs. Row-Stores: How Different Are They Really?” Abadi, Madden, and Hachem. SIGMOD 2008.

Queries touch 3-4 foreign keys in fact table, 1-2 numeric columns
Complete fact table takes up ~4 GB (compressed)
Vertically partitioned tables take up 0.7-1.1 GB (compressed)
What’s Going On? All Indexes Case

1. Tuple construction
   1. Common type of query: 
      
      ```sql
      SELECT store_name, SUM(revenue) 
      FROM Facts, Stores 
      WHERE fact.store_id = stores.store_id 
      AND stores.country = “Canada” 
      GROUP BY store_name
      ```

   1. Result of lower part of query plan is a set of TIDs that passed all predicates

   1. Need to extract SELECT attributes at these TIDs
      1. BUT: index maps value to TID
      1. You really want to map TID to value (i.e., a vertical partition)

 Tuple construction is SLOW
So….

1. All indexes approach is a poor way to simulate a column-store
2. Problems with vertical partitioning are NOT fundamental
   1. Store tuple header in a separate partition
   1. Allow virtual TIDs
   1. Combine clustered indexes, vertical partitioning
3. So can row-stores simulate column-stores?
   1. Might be possible, BUT:
      1. Need better support for vertical partitioning at the storage layer
      1. Need support for column-specific optimizations at the executer level
      1. Full integration: buffer pool, transaction manager, ..
4. When will this happen?
   1. Most promising features = soon
5. ..unless new technology / new objectives change the game (SSDs, Massively Parallel Platforms, Energy-efficiency)
End of Part 1

1 Basic concepts — Stavros
   1 Introduction to key features
   1 From DSM to column-stores and performance tradeoffs
   1 Column-store architecture overview
   1 Will rows and columns ever converge?

Part 2: Column-oriented execution — Daniel

Part 3: MonetDB/X100 and CPU efficiency — Peter