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

<table>
<thead>
<tr>
<th>Date</th>
<th>Store</th>
<th>Product</th>
<th>Customer</th>
<th>Price</th>
</tr>
</thead>
</table>

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

### column-store

<table>
<thead>
<tr>
<th>Date</th>
<th>Store</th>
<th>Product</th>
<th>Customer</th>
<th>Price</th>
</tr>
</thead>
</table>

- 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?

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

- How did we get here, and where we are heading
- What are the column-specific optimizations?
- How do we improve CPU efficiency when operating on Cs

Part 1

Part 2

Part 3
Outline

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

● Part 2: Column-oriented execution — Daniel

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

- Typical DW installation
- Real-world example

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

QUERY 2
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

caveats:

• “select * ” not any faster
• clever disk prefetching
• clever tuple reconstruction

What about vertical partitioning? (it does not work with ad-hoc queries)
Telco example explained (2/3): compression efficiency

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

- Why?
  - Rows contain values from different domains
    => more entropy, difficult to dense-pack
  - Columns exhibit significantly less entropy
  - Examples:
    Male, Female, Female, Female, Male

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

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

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

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

- **Exploit columnar storage in other DBMS components**
  - Physical design (both static and dynamic)
Effect on C-Store performance

Average for SSBM queries on C-store

Time (sec)

0 10 20 30 40 50

original C-store

column-oriented join algorithm

enable compression & operate on compressed

enable late materialization
Summary of column-store key features

- Storage layout
  - Columnar storage
  - Header/ID elimination
  - Compression
  - Multiple sort orders

- Execution engine
  - Column operators
  - Avoid decompression
  - Late materialization
  - Vectorized operations

- Design tools, optimizer
Outline

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

- Part 2: Column-oriented execution — Daniel

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

70s -1985:

- DSM paper
- "A decomposition storage model"
  Copeland and Khoshafian. SIGMOD 1985

1985: DSM paper

- "A Modular, Self-Describing Clinical Databank System,"
  Computers and Biomedical Research, 1975
- "An overview of cantor: a new system for data analysis”
  Karasalo, Svensson, SSDBM 1983

1990s: Commercialization through SybaseIQ

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

  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


● PAX: Partition Attributes Across
  ● Retains NSM I/O pattern
  ● Optimizes cache-to-RAM communication


More hybrid NSM/DSM schemes

- **Dynamic PAX: Data Morphing**
  

- **Clotho: custom layout using scatter-gather I/O**
  

- **Fractured mirrors**
  
  - 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

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

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

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

- **MonetDB/X100**
  - “proper” disk-based column store

- **Explosion of new products**
Performance tradeoffs: columns vs. rows

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

- Optimized DSM in “Fractured Mirrors,” 2002
- “Apples-to-apples” comparison
- Follow-up study
- Main-memory DSM vs. NSM
- Flash-disks: a come-back for PAX?

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

“Read-Optimized Databases, In-Depth” Holloway, DeWitt,
VLDB’06

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

“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

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

Similar: storage density comparable to column stores


Fractured mirrors: performance

From PAX paper:

- Chunk-based tuple merging
  - Read in segments of M pages
  - Merge segments in memory
  - Becomes CPU-bound after 5 pages

![Graph showing performance of NSM, PAX, and DSM](image)

- regular DSM

![Diagram with time and columns projected](image)

- From PAX paper:
  - Optimized DSM
Column-scanner implementation

Row scanner

Column scanner

SELECT name, age WHERE age > 40

apply predicate(s)

Direct I/O

prefetch ~100ms worth of data

1 Joe 45
2 Sue 37
… ...

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

Joe 45
… ...

45 37 ...

#POS

apply predicate #1
Scan performance

- Large prefetch hides disk seeks in columns
- Column-CPU efficiency with lower selectivity
- Row-CPU suffers from memory stalls
- Memory stalls disappear in narrow tuples
- 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

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

Graph showing the relationship between time (sec) and selected bytes per tuple for different columns with and without disk traffic.
Varying prefetch size

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

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

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

![Graphs showing CPU performance trade-offs](image)

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

- Performance characteristics
  - very fast random reads, slow random writes
  - fast sequential reads and writes
- Price per bit (capacity follows)
  - cheaper than RAM, order of magnitude more expensive than Disk
- Flash Translation Layer introduces unpredictability
  - avoid random writes!
- Form factors not ideal yet
  - SSD (⇒ small reads still suffer from SATA overhead/OS limitations)
  - PCI card (⇒ high price, limited expandability)

- Boost Sequential I/O in a simple package
  - Flash RAID: very tight bandwidth/cm³ packing (4GB/sec inside the box)
- Column Store Updates
  - useful for delta structures and logs
- 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
- 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

- New-generation SSDs
  - Very fast random reads, slower random writes
  - Fast sequential RW, comparable to HDD arrays
- No expensive seeks across columns
- 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 to 1.2x slower to same
- column-store (2006) ..to 2x slower ..to same
- optimized DSM (2002) and 3x faster!
- SSD Postgres/PAX (2009)
Outline

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

- Part 2: Column-oriented execution — Daniel

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

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

engine
- block-tuple operators
- new access methods
- optimized relational operators

system-level
- system-wide column support
- loading / updates
- scaling through multiple nodes
- transactions / redundancy
C-Store

- Compress columns
- No alignment
- Big disk blocks
- Only materialized views (perhaps many)
- Focus on Sorting not indexing
- Data ordered on anything, not just time
- Automatic physical DBMS design
- Optimize for grid computing
- Innovative redundancy
- Xacts – but no need for Mohan
- Column optimizer and executor

C-Store: only materialized views (MV$s$)

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

**User view:**
- EMP (name, age, salary, dept)
- Dept (dname, floor)

**Possible set of MV$s$:**
- MV-1 (name, dept, floor) in floor order
- MV-2 (salary, age) in age order
- MV-3 (dname, salary, name) in salary order
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 direct

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
Applications for column-stores

- Data Warehousing
  - High end (clustering)
  - Mid end/Mass Market
  - Personal Analytics
- Data Mining
  - E.g. Proximity
- Google BigTable
- RDF
  - Semantic web data management
- Information retrieval
  - Terabyte TREC
- Scientific datasets
  - SciDB initiative
  - SLOAN Digital Sky Survey on MonetDB
List of column-store systems

- Cantor (history)
- Sybase IQ
- SenSage (former Addamark Technologies)
- Kdb
- 1010data
- MonetDB
- C-Store/Vertica
- X100/VectorWise
- KickFire
- SAP Business Accelerator
- Infobright
- ParAccel
- Exasol
Outline

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

- Part 2: Column-oriented execution — Daniel

- 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 | 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

```
<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>
```

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>
```

Option B: Index Every Column

```
<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

Experiments

- Star Schema Benchmark (SSBM)
  - Implemented by professional DBA
  - Original row-store plus 2 column-store simulations on same row-store product

<table>
<thead>
<tr>
<th></th>
<th>Normal Row-Store</th>
<th>Vertically Partitioned Row-Store</th>
<th>Row-Store With All Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Average</strong></td>
<td>25.7</td>
<td>79.9</td>
<td>221.2</td>
</tr>
</tbody>
</table>

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

“Column-Stores vs Row-Stores: How Different are They Really?” Abadi, Hachem, and Madden. SIGMOD 2008.
What’s Going On? Vertical Partitions

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

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

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

- Tuple construction
  - Common type of query:
    ```
    SELECT store_name, SUM(revenue)
    FROM Facts, Stores
    WHERE fact.store_id = stores.store_id
    AND stores.country = “Canada”
    GROUP BY store_name
    ```
  - Result of lower part of query plan is a set of TIDs that passed all predicates
  - Need to extract SELECT attributes at these TIDs
    - BUT: index maps value to TID
    - You really want to map TID to value (i.e., a vertical partition)
  - Tuple construction is SLOW
So....

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

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

- Part 2: Column-oriented execution — Daniel

- Part 3: MonetDB/X100 and CPU efficiency — Peter
Part 2 Outline

- Compression
- Tuple Materialization
- Joins
Column-Oriented Database Systems

Compression

“Super-Scalar RAM-CPU Cache Compression”
Zukowski, Heman, Nes, Boncz, ICDE’06

“Integrating Compression and Execution in Column-Oriented Database Systems”
Abadi, Madden, and Ferreira, SIGMOD ’06

• Query optimization in compressed database systems”
Chen, Gehrke, Korn, SIGMOD’01
Compression

- Trades I/O for CPU
- Increased column-store opportunities:
  - Higher data value locality in column stores
  - Techniques such as run length encoding far more useful
  - Can use extra space to store multiple copies of data in different sort orders
### Run-length Encoding

<table>
<thead>
<tr>
<th>Quarter</th>
<th>Product ID</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Q1</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Q1</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>Q1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Q1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Q1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Q2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Q2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Q2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Q2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

### Quarter Encoding

- (Q1, 1, 300)
- (Q2, 301, 350)
- (Q3, 651, 500)
- (Q4, 1151, 600)

### Product ID Encoding

- (1, 1, 5)
- (2, 304, 1)
- (1, 301, 3)

### Price Encoding

- 5
- 7
- 2
- 6
- 8
- 5
- 3
- 8
- 1
- 4
- 3
- 8
- 1
- 4
Bit-vector Encoding

- For each unique value, $v$, in column $c$, create bit-vector $b$
  - $b[i] = 1$ if $c[i] = v$
- Good for columns with few unique values
- Each bit-vector can be further compressed if sparse

<table>
<thead>
<tr>
<th>Product ID</th>
<th>ID: 1</th>
<th>ID: 2</th>
<th>ID: 3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Dictionary Encoding

- For each unique value create dictionary entry
- Dictionary can be per-block or per-column
- Column-stores have the advantage that dictionary entries may encode multiple values at once

```
Quarter
Q1 1
Q2 3
Q4 0
Q1 2
Q3 0
Q1 0
Q2 3
Q4 2
Q3 2
...
```

```
Quarter
0 24
1 128
2 122
3 3
```

```
Dictionary
0: Q1
1: Q2
2: Q3
3: Q4
...
```

```
Dictionary
24: Q1, Q2, Q4, Q1
122: Q2, Q4, Q3, Q3
128: Q3, Q1, Q1, Q1
```
## Frame Of Reference Encoding

- **Encodes values as b bit offset from chosen frame of reference**
- **Special escape code (e.g. all bits set to 1) indicates a difference larger than can be stored in b bits**
  - After escape code, original (uncompressed) value is written

### Example

<table>
<thead>
<tr>
<th>Price</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>-5</td>
</tr>
<tr>
<td>54</td>
<td>4</td>
</tr>
<tr>
<td>48</td>
<td>-2</td>
</tr>
<tr>
<td>55</td>
<td>5</td>
</tr>
<tr>
<td>51</td>
<td>1</td>
</tr>
<tr>
<td>53</td>
<td>3</td>
</tr>
<tr>
<td>40</td>
<td>∞</td>
</tr>
<tr>
<td>50</td>
<td>40</td>
</tr>
<tr>
<td>49</td>
<td>0</td>
</tr>
<tr>
<td>62</td>
<td>-1</td>
</tr>
<tr>
<td>52</td>
<td>∞</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
</tr>
</tbody>
</table>

**4 bits per value**

**Exceptions (see part 3 for a better way to deal with exceptions)**

"Compressing Relations and Indexes " Goldstein, Ramakrishnan, Shaft, ICDE’ 98
Differential Encoding

- Encodes values as b bit offset from previous value
- Special escape code (just like frame of reference encoding) indicates a difference larger than can be stored in b bits
  - After escape code, original (uncompressed) value is written
- Performs well on columns containing increasing/decreasing sequences
  - inverted lists
  - timestamps
  - object IDs
  - sorted / clustered columns

“Improved Word-Aligned Binary Compression for Text Indexing”
Ahn, Moffat, TKDE’ 06
What Compression Scheme To Use?

Does column appear in the sort key?

- Yes
  - Is the average run-length > 2?
    - Yes: RLE
    - No: Differential Encoding
  - No: Are number of unique values < ~50000?
    - Yes: Does this column appear frequently in selection predicates?
      - Yes: Bit-vector Compression
      - No: Dictionary Compression
    - No: Heavyweight Compression

- No: Is the data numerical and exhibit good locality?
  - Yes: Frame of Reference Encoding
  - No: Leave Data Uncompressed
Heavy-Weight Compression Schemes

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Decompression Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>BZIP</td>
<td>10 MB/s</td>
</tr>
<tr>
<td>ZLIB</td>
<td>80 MB/s</td>
</tr>
<tr>
<td>LZO</td>
<td>300 MB/s</td>
</tr>
</tbody>
</table>

- Modern disk arrays can achieve > 1GB/s
- 1/3 CPU for decompression ⇒ 3GB/s needed

⇒ Lightweight compression schemes are better
⇒ Even better: operate directly on compressed data

“Super-Scalar RAM-CPU Cache Compression”
Zukowski, Heman, Nes, Boncz, ICDE’06
Operating Directly on Compressed Data

- I/O - CPU tradeoff is no longer a tradeoff
- Reduces memory–CPU bandwidth requirements
- Opens up possibility of operating on multiple records at once
Operating Directly on Compressed Data

<table>
<thead>
<tr>
<th>Quarter</th>
<th>Product ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Q1, 1, 300)</td>
<td>1 0 0</td>
</tr>
<tr>
<td>(Q2, 301, 6)</td>
<td>0 0 1</td>
</tr>
<tr>
<td>(Q3, 307, 500)</td>
<td>1 0 0</td>
</tr>
<tr>
<td>(Q4, 807, 600)</td>
<td>0 1 0</td>
</tr>
</tbody>
</table>

```
SELECT ProductID, COUNT(*)
FROM table
WHERE (Quarter = Q2)
GROUP BY ProductID
```
Operating Directly on Compressed Data

**Block API**

- `Data`
  - `isOneValue()`
  - `isValueSorted()`
  - `isPosContiguous()`
  - `isSparse()`
  - `getNext()`
  - `decompressIntoArray()`
  - `getValueAtPosition(pos)`
  - `getMin()`
  - `getMax()`
  - `getSize()`

**SELECT**

```sql
SELECT ProductID, Count(*)
FROM table
WHERE (Quarter = Q2)
GROUP BY ProductID
```

<table>
<thead>
<tr>
<th>Quarter</th>
<th>ProductID</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1</td>
<td>300</td>
</tr>
<tr>
<td>Q2</td>
<td>301</td>
<td>6</td>
</tr>
<tr>
<td>Q3</td>
<td>307</td>
<td>500</td>
</tr>
<tr>
<td>Q4</td>
<td>807</td>
<td>600</td>
</tr>
</tbody>
</table>

"Integrating Compression and Execution in Column-Oriented Database Systems" Abadi et. al, SIGMOD ’06
Column-Oriented Database Systems

Tuple Materialization and Column-Oriented Join Algorithms


“Self-organizing tuple reconstruction in column-stores”, Idreos, Manegold, Kersten, SIGMOD’ 09

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


“Cache-Conscious Radix-Decluster Projections”, Manegold, Boncz, Nes, VLDB’ 04
When should columns be projected?

- Where should column projection operators be placed in a query plan?
  - **Row-store:**
    - Column projection involves removing unneeded columns from tuples
    - Generally done as early as possible
  - **Column-store:**
    - Operation is almost completely opposite from a row-store
    - Column projection involves reading needed columns from storage and extracting values for a listed set of tuples
      - This process is called “materialization”
    - Early materialization: project columns at beginning of query plan
      - Straightforward since there is a one-to-one mapping across columns
    - Late materialization: wait as long as possible for projecting columns
      - More complicated since selection and join operators on one column obfuscates mapping to other columns from same table
  - Most column-stores construct tuples and column projection time
    - Many database interfaces expect output in regular tuples (rows)
    - Rest of discussion will focus on this case
When should tuples be constructed?

Solution 1: Create rows first (EM).

But:
- Need to construct ALL tuples
- Need to decompress data
- Poor memory bandwidth utilization

QUERY:
```
SELECT custID, SUM(price)
FROM table
WHERE (prodID = 4) AND (storeID = 1) AND
GROUP BY custID
```
Solution 2: Operate on columns

QUERY:
SELECT custID, SUM(price)
FROM table
WHERE (prodID = 4) AND (storeID = 1) AND
GROUP BY custID
Solution 2: Operate on columns

QUERY:
SELECT custID, SUM(price) FROM table
WHERE (prodID = 4) AND (storeID = 1) AND GROUP BY custID
Solution 2: Operate on columns

QUERY:
SELECT custID, SUM(price)
FROM table
WHERE (prodID = 4) AND (storeID = 1) AND
GROUP BY custID

Data Source
custID
prodID
storeID
price

AGG
AND

0 1
0 1
0 1
0 1
0 1

3 3
13 80
7 13
42 80
2 3
3 3
3 3
2 3
3 3
3 3

prodID
storeID
custID
price
Solution 2: Operate on columns

QUERY:
SELECT custID, SUM(price)
FROM table
WHERE (prodID = 4) AND (storeID = 1) AND
GROUP BY custID
For plans without joins, late materialization is a win

**QUERY:**

```sql
SELECT C₁, SUM(C₂)
FROM table
WHERE (C₁ < CONST) AND (C₂ < CONST)
GROUP BY C₁
```

- Ran on 2 compressed columns from TPC-H scale 10 data
Even on uncompressed data, late materialization is still a win

**QUERY:**

```sql
SELECT C_1, SUM(C_2)
FROM table
WHERE (C_1 < CONST) AND (C_2 < CONST)
GROUP BY C_1
```

- Materializing late still works best
What about for plans with joins?

From R1, R2, R3
What about for plans with joins?

From R1, R2, R3
Early Materialization Example

Construct

Facts

Customers

(4,1,4)
(1,2,3)
(1,2,3)
(1,2,3)

prodID   storeID   quantity   custID   price
12       1          6           2         7
11       1          2           3         13
1         1          1           3         42
1         1          1           3         80

QUERY:
SELECT C.lastName, SUM(F.price)
FROM facts AS F, customers AS C
WHERE F.custID = C.custID
GROUP BY C.lastName

1  Green
2  White
3  Brown

1  Green
2  White
3  Brown
Early Materialization Example

QUERY:
```
SELECT C.lastName, SUM(F.price) 
FROM facts AS F, customers AS C 
WHERE F.custID = C.custID 
GROUP BY C.lastName
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>White</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Brown</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Brown</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>Brown</td>
<td></td>
</tr>
</tbody>
</table>

Join

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Green</td>
</tr>
<tr>
<td>2</td>
<td>White</td>
</tr>
<tr>
<td>3</td>
<td>Brown</td>
</tr>
</tbody>
</table>
Late Materialization Example

**QUERY:**

```sql
SELECT C.lastName, SUM(F.price)
FROM facts AS F, customers AS C
WHERE F.custID = C.custID
GROUP BY C.lastName
```

Late materialized join causes out of order probing of projected columns from the inner relation.
Late Materialized Join Performance

- Naïve LM join about 2X slower than EM join on typical queries (due to random I/O)
  - This number is very dependent on
    - Amount of memory available
    - Number of projected attributes
    - Join cardinality

- But we can do better
  - Invisible Join
  - Jive/Flash Join
  - Radix cluster/decluster join
Invisible Join

- Designed for typical joins when data is modeled using a star schema
  - One ("fact") table is joined with multiple dimension tables
- Typical query:
  ```sql
  select c_nation, s_nation, d_year,
         sum(lo_revenue) as revenue
  from customer, lineorder, supplier, date
  where lo_custkey = c_custkey
  and lo_suppkey = s_suppkey
  and lo_orderdate = d_datekey
  and c_region = 'ASIA'
  and s_region = 'ASIA'
  and d_year >= 1992 and d_year <= 1997
  group by c_nation, s_nation, d_year
  order by d_year asc, revenue desc;
  ```
### Invisible Join

#### Apply “region = ‘Asia’” On Customer Table

<table>
<thead>
<tr>
<th>custkey</th>
<th>region</th>
<th>nation</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ASIA</td>
<td>CHINA</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>ASIA</td>
<td>INDIA</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>ASIA</td>
<td>INDIA</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>EUROPE</td>
<td>FRANCE</td>
<td>...</td>
</tr>
</tbody>
</table>

Hash Table (or bit-map) Containing Keys 1, 2 and 3

#### Apply “region = ‘Asia’” On Supplier Table

<table>
<thead>
<tr>
<th>suppkey</th>
<th>region</th>
<th>nation</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ASIA</td>
<td>RUSSIA</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>EUROPE</td>
<td>SPAIN</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>ASIA</td>
<td>JAPAN</td>
<td>...</td>
</tr>
</tbody>
</table>

Hash Table (or bit-map) Containing Keys 1, 3

#### Apply “year in [1992,1997]” On Date Table

<table>
<thead>
<tr>
<th>dateid</th>
<th>year</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>01011997</td>
<td>1997</td>
<td></td>
</tr>
<tr>
<td>01021997</td>
<td>1997</td>
<td></td>
</tr>
<tr>
<td>01031997</td>
<td>1997</td>
<td></td>
</tr>
</tbody>
</table>

Hash Table Containing Keys 01011997, 01021997, and 01031997
Column-Stores vs Row-Stores: How Different are They Really?
Abadi et. al. SIGMOD 2008

Original Fact Table

<table>
<thead>
<tr>
<th>orderkey</th>
<th>custkey</th>
<th>suppkey</th>
<th>orderdate</th>
<th>revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>01011997</td>
<td>43256</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>01011997</td>
<td>33333</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>01021997</td>
<td>12121</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>01021997</td>
<td>23233</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>2</td>
<td>01021997</td>
<td>45456</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>01031997</td>
<td>43251</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>2</td>
<td>01031997</td>
<td>34235</td>
</tr>
</tbody>
</table>

Hash Table Containing Keys 1, 2 and 3

Hash Table Containing Keys 1 and 3

Hash Table Containing Keys 01011997, 01021997, and 01031997
**Invisible Join**

Apply “region = ‘Asia’” On Customer Table

<table>
<thead>
<tr>
<th>custkey</th>
<th>region</th>
<th>nation</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ASIA</td>
<td>CHINA</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>ASIA</td>
<td>INDIA</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>ASIA</td>
<td>INDIA</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>EUROPE</td>
<td>FRANCE</td>
<td>...</td>
</tr>
</tbody>
</table>

Apply “region = ‘Asia’” On Supplier Table

<table>
<thead>
<tr>
<th>suppkey</th>
<th>region</th>
<th>nation</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ASIA</td>
<td>RUSSIA</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>EUROPE</td>
<td>SPAIN</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>ASIA</td>
<td>JAPAN</td>
<td>...</td>
</tr>
</tbody>
</table>

Apply “year in [1992,1997]” On Date Table

<table>
<thead>
<tr>
<th>dateid</th>
<th>year</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>01011997</td>
<td>1997</td>
<td>...</td>
</tr>
<tr>
<td>01021997</td>
<td>1997</td>
<td>...</td>
</tr>
<tr>
<td>01031997</td>
<td>1997</td>
<td>...</td>
</tr>
</tbody>
</table>

Hash Table (or bit-map) Containing Keys 1, 2 and 3

Hash Table (or bit-map) Containing Keys 1, 3

Hash Table Containing Keys 01011997, 01021997, and 01031997

---

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

- **Bottom Line**
  - Many data warehouses model data using star/snowflake schemes
  - Joins of one (fact) table with many dimension tables is common
  - Invisible join takes advantage of this by making sure that the table that can be accessed in position order is the fact table for each join
  - Position lists from the fact table are then intersected (in position order)
  - This reduces the amount of data that must be accessed out of order from the dimension tables
  - “Between-predicate rewriting” trick not relevant for this discussion
Still accessing table out of order
Jive/Flash Join


Still accessing table out of order
Jive/Flash Join

1. Add column with dense ascending integers from 1

2. Sort new position list by second column

3. Probe projected column in order using new sorted position list, keeping first column from position list around

4. Sort new result by first column
Jive/Flash Join

● Bottom Line
  ● Instead of probing projected columns from inner table out of order:
    ● Sort join index
    ● Probe projected columns in order
    ● Sort result using an added column
  
  ● LM vs EM tradeoffs:
    ● LM has the extra sorts (EM accesses all columns in order)
    ● LM only has to fit join columns into memory (EM needs join columns and all projected columns)
      ▪ Results in big memory and CPU savings (see part 3 for why there is CPU savings)
    ● LM only has to materialize relevant columns
    ● In many cases LM advantages outweigh disadvantages
  
  ● LM would be a clear winner if not for those pesky sorts … can we do better?
Radix Cluster/Decluster

- The full sort from the Jive join is actually overkill
  - We just want to access the storage blocks in order (we don’t mind random access within a block)
  - So do a radix sort and stop early
  - By stopping early, data within each block is accessed out of order, but in the order specified in the original join index
  - Use this pseudo-order to accelerate the post-probe sort as well

- “Generic Database Cost Models for Hierarchical Memory Systems”, VLDB’ 02 (all Manegold, Boncz, Kersten)
Radix Cluster/Decluster

- **Bottom line**
  - Both sorts from the Jive join can be significantly reduced in overhead
  - Only been tested when there is sufficient memory for the entire join index to be stored three times
    - Technique is likely applicable to larger join indexes, but utility will go down a little
  - Only works if random access within a storage block
    - Don’t want to use radix cluster/decluster if you have variable-width column values or compression schemes that can only be decompressed starting from the beginning of the block
LM vs EM joins

- Invisible, Jive, Flash, Cluster, Decluster techniques contain a bag of tricks to improve LM joins
- Research papers show that LM joins become 2X faster than EM joins (instead of 2X slower) for a wide array of query types
Tuple Construction Heuristics

- For queries with selective predicates, aggregations, or compressed data, use late materialization
- For joins:
  - Research papers:
    - Always use late materialization
  - Commercial systems:
    - Inner table to a join often materialized before join (reduces system complexity):
    - Some systems will use LM only if columns from inner table can fit entirely in memory
Outline

- Computational Efficiency of DB on modern hardware
  - how column-stores can help here
  - Keynote revisited: MonetDB & VectorWise in more depth

- CPU efficient column compression
  - vectorized decompression

- Conclusions
  - future work
Column-Oriented Database Systems

40 years of hardware evolution vs. DBMS computational efficiency
CPU Architecture

Elements:

- Storage
  - CPU caches L1/L2/L3
- Registers
- Execution Unit(s)
  - Pipelined
  - SIMD
## CPU Metrics

<table>
<thead>
<tr>
<th>Processor</th>
<th>16-bit address/bus, micro-coded</th>
<th>32-bit address/bus, micro-coded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td>80286</td>
<td>80386</td>
</tr>
<tr>
<td>Year</td>
<td>1982</td>
<td>1990</td>
</tr>
<tr>
<td>Transistors (thousands)</td>
<td>134</td>
<td>220</td>
</tr>
<tr>
<td>Latency (clocks)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Bus width (bits)</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Clock rate (MHz)</td>
<td>12.5</td>
<td>33</td>
</tr>
<tr>
<td>Bandwidth (MIPS)</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Latency (ns)</td>
<td>320</td>
<td>380</td>
</tr>
</tbody>
</table>
## CPU Metrics

<table>
<thead>
<tr>
<th>Processor</th>
<th>16-bit address/bus, micro-coded</th>
<th>32-bit address/bus, micro-coded</th>
<th>5-stage pipeline, on-chip I&amp;D caches, FPU</th>
<th>2-way superscalar, 64-bit bus</th>
<th>Out-of-order, 3-way superscalar</th>
<th>Out-of-order, super-pipelined, on-chip L2 cache</th>
<th>Multi-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td>80286</td>
<td>80386</td>
<td>80486</td>
<td>Pentium</td>
<td>PentiumPro</td>
<td>Pentium4</td>
<td>CoreDuo</td>
</tr>
<tr>
<td>Transistors (thousands)</td>
<td>134</td>
<td>275</td>
<td>1,200</td>
<td>3,100</td>
<td>5,500</td>
<td>42,000</td>
<td>151,600</td>
</tr>
<tr>
<td>Latency (clocks)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>22</td>
<td>12</td>
</tr>
<tr>
<td>Bus width (bits)</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Clock rate (MHz)</td>
<td>12.5</td>
<td>16</td>
<td>25</td>
<td>66</td>
<td>200</td>
<td>1500</td>
<td>2333</td>
</tr>
<tr>
<td>Bandwidth (MIPS)</td>
<td>2</td>
<td>6</td>
<td>25</td>
<td>132</td>
<td>600</td>
<td>4500</td>
<td>21000</td>
</tr>
<tr>
<td>Latency (ns)</td>
<td>320</td>
<td>313</td>
<td>200</td>
<td>76</td>
<td>50</td>
<td>15</td>
<td>5</td>
</tr>
</tbody>
</table>
DRAM Metrics

![Graph showing DRAM metrics over time]

- Capacity
- Bandwidth
- Random access (1/latency)

Years: '80, '83, '86, '93, '97, '00, '06
Super-Scalar Execution (pipelining)

Sequential execution

Instruction fetch
Instruction decode
Execute
Write back

IF-1
ID-1
EX-1
WB-1

IF-2
ID-2
IE-2
WB-2

IF-3
ID-3
EX-3
WB-3

CPU cycle

Pipelined execution

Instruction fetch
Instruction decode
Execute
Write back

IF-1 IF-2 IF-3 IF-4 IF-5 IF-6
ID-1 ID-2 ID-3 ID-4 ID-5
EX-1 EX-2 EX-3 EX-4
WB-1 WB-2 WB-3

CPU cycle

Time
Hazards

- **Data hazards**
  - Dependencies between instructions
  - L1 data cache misses

- **Control Hazards**
  - Branch mispredictions
  - Computed branches (late binding)
  - L1 instruction cache misses

Result: bubbles in the pipeline

Out-of-order execution addresses data hazards
- control hazards typically more expensive
SIMD

- Single Instruction Multiple Data
  - Same operation applied on a vector of values
  - MMX: 64 bits, SSE: 128bits, AVX: 256bits
  - SSE, e.g. multiply 8 short integers
A Look at the Query Pipeline

```
SELECT id, name
   (age-30)*50 AS bonus
FROM employee
WHERE age > 30
```
A Look at the Query Pipeline

Operators

Iterator interface
- `open()`
- `next()`: tuple
- `close()`
A Look at the Query Pipeline

Primitives

Provide computational functionality

All arithmetic allowed in expressions, e.g. Multiplication

mult(int,int) \rightarrow\ int
Database Architecture causes Hazards

DB workload execution on a modern computer

Processor

BUSY

IDLE (STALLED)

100%

80%

60%

40%

20%

0%

Ideal

seq. scan

index scan

DSS

OLTP

“DBMSs On A Modern Processor: Where Does Time Go?”
Ailamaki, DeWitt, Hill, Wood, VLDB’ 99
DBMS Computational Efficiency

TPC-H 1GB, query 1

- selects 98% of fact table, computes net prices and aggregates all

Results:

- C program: ?
- MySQL: 26.2s
- DBMS “X”: 28.1s

“MonetDB/X100: Hyper-Pipelining Query Execution” Boncz, Zukowski, Nes, CIDR’05
DBMS Computational Efficiency

TPC-H 1GB, query 1

- selects 98% of fact table, computes net prices and aggregates all

- Results:
  - C program: 0.2s
  - MySQL: 26.2s
  - DBMS "X": 28.1s

“MonetDB/X100: Hyper-Pipelining Query Execution” Boncz, Zukowski, Nes, CIDR ’05
Column-Oriented Database Systems
a column-store

- "save disk I/O when scan-intensive queries need a few columns"
- "avoid an expression interpreter to improve computational efficiency"
### RISC Database Algebra

```
<table>
<thead>
<tr>
<th>people_id</th>
<th>people_name</th>
<th>people_age</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>102</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>104</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>105</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>108</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>109</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>112</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>113</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>114</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>115</td>
<td>9</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>select(30,nil)</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>
<tr>
<td>9 35</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>tmp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 7</td>
</tr>
<tr>
<td>1 15</td>
</tr>
<tr>
<td>2 1</td>
</tr>
<tr>
<td>3 12</td>
</tr>
<tr>
<td>4 5</td>
</tr>
</tbody>
</table>
```

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

CPU happy? 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 Database Algebra

CPU happy? 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;
}
```

MATERIALIZED intermediate results
a column-store

- “save disk I/O when scan-intensive queries need a few columns”
- “avoid an expression interpreter to improve computational efficiency”

How?

- RISC query algebra: hard-coded semantics
  - Decompose complex expressions in multiple operations
- Operators only handle **simple arrays**
  - No code that handles slotted buffered record layout
- Relational algebra becomes **array manipulation language**
  - Often SIMD for free
- Plus: use of **cache-conscious** algorithms for Sort/Aggr/Join
You want efficiency
  - Simple hard-coded operators

I take scalability
  - Result materialization

<table>
<thead>
<tr>
<th>System</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>C program</td>
<td>0.2s</td>
</tr>
<tr>
<td>MonetDB</td>
<td>3.7s</td>
</tr>
<tr>
<td>MySQL</td>
<td>26.2s</td>
</tr>
<tr>
<td>DBMS “X”</td>
<td>28.1s</td>
</tr>
</tbody>
</table>
Column-Oriented Database Systems

as a research platform
SIGMOD 1985

MonetDB supports SQL, XML, ODMG, ...RDF

RDF support on C-STORE / SW-Store

“MIL Primitives for Querying a Fragmented World”, Boncz, Kersten, VLDBJ’ 98
“Flattening an Object Algebra to Provide Performance” Boncz, Wilschut, Kersten, ICDE’ 98
“MonetDB/XQuery: a fast XQuery processor powered by a relational engine” Boncz, Grust, vanKeulen, Rittinger, Teubner, SIGMOD’ 06
“SW-Store: a vertically partitioned DBMS for Semantic Web data management“ Abadi, Marcus, Madden, Hollenbach, VLDBJ’ 09
Cache-Conscious Joins
- Cost Models, Radix-cluster, Radix-decluster

MonetDB/XQuery:
- structural joins exploiting positional column access

Cracking:
- on-the-fly automatic indexing without workload knowledge

Recycling:
- using materialized intermediates

Run-time Query Optimization:
- correlation-aware run-time optimization without cost model

MonetDB/XQuery: a fast XQuery processor powered by a relational engine, Boncz, Grust, vanKeulen, Rittinger, Teubner, SIGMOD’06

“Generic Database Cost Models for Hierarchical Memory Systems”, VLDB’02 (all Manegold, Boncz, Kersten)
“Cache-Conscious Radix-Decluster Projections”, Manegold, Boncz, Nes, VLDB’04

“Database Cracking”, CIDR’07
“Updating a cracked database”, SIGMOD’07
“Self-organizing tuple reconstruction in column-stores”, SIGMOD’09 (all Idreos, Manegold, Kersten)

“An architecture for recycling intermediates in a column-store”, Ivanova, Kersten, Nes, Goncalves, SIGMOD’09

“ROX: run-time optimization of XQueries”, Abdelkader, Boncz, Manegold, vanKeulen, SIGMOD’09
Column-Oriented Database Systems

"MonetDB/X100"

vectorized query processing
MonetDB spin-off: MonetDB/X100

Materialization vs Pipelining
“MonetDB/X100: Hyper-Pipelining Query Execution” Boncz, Zukowski, Nes, CIDR’05
"MonetDB/X100: Hyper-Pipelining Query Execution" Boncz, Zukowski, Nes, CIDR’05
“Vectorized In Cache Processing”

vector = array of ~100

processed in a tight loop

CPU cache Resident

“MonetDB/X100: Hyper-Pipelining Query Execution” Boncz, Zukowski, Nes, CIDR’05
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 Efficiency depends on “nice” code**
- out-of-order execution
- few dependencies (control, data)
- compiler support

**Compilers like simple loops over arrays**
- loop-pipelining
- automatic SIMD
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 Efficiency depends on “nice” code
- out-of-order execution
- few dependencies (control,data)
- compiler support

Compilers like simple loops over arrays
- loop-pipelining
- automatic SIMD

```
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)
```
Tricks being played:

- Late materialization

- Materialization avoidance using selection vectors
map_mul_flt_val_flt_col(
    float *res,
    int*  sel,
    float  val,
    float *col, int n)
{
    for(int i=0; i<n; i++)
        res[i] = val * col[sel[i]];
}

**selection vectors** used to reduce vector copying

contain selected positions
map_mul_flt_val_flt_col(  
    float *res,  
    int* sel,  
    float val,  
    float *col, int n)  
{
    for(int i=0; i<n; i++)
        res[i] = val * col[sel[i]];  
}

selection vectors used to reduce vector copying

contain selected positions

“MonetDB/X100: Hyper-Pipelining Query Execution ” Boncz, Zukowski, Nes, CIDR’ 05
MonetDB/X100

- Both efficiency
  - Vectorized primitives
- and scalability..
  - Pipelined query evaluation

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>C program</td>
<td>0.2s</td>
</tr>
<tr>
<td>MonetDB/X100</td>
<td>0.6s</td>
</tr>
<tr>
<td>MonetDB</td>
<td>3.7s</td>
</tr>
<tr>
<td>MySQL</td>
<td>26.2s</td>
</tr>
<tr>
<td>DBMS “X”</td>
<td>28.1s</td>
</tr>
</tbody>
</table>
“MonetDB/X100: Hyper-Pipelining Query Execution” Boncz, Zukowski, Nes, CIDR’05

Memory Hierarchy

CPU cache

ColumnBM (buffer manager)

X100 query engine

SELECT

PROJECT

SCAN

CPU

Main Memory

Harddrive

Small
Fast
Expensive

~10 GB/s
2–20 cycles

2–3 GB/s
150–250 cycles

40–400 MB/s
millions of cycles

Large
Slow
Cheap

MonetDB/X100: Hyper-Pipelining Query Execution
Boncz, Zukowski, Nes, CIDR’05
Memory Hierarchy

Vectors are only the in-cache representation

RAM & disk representation might actually be different

(vectorwise uses both PAX & DSM)
Optimal Vector size?

All vectors together should fit the CPU cache

Optimizer should tune this, given the query characteristics.

“MonetDB/X100: Hyper-Pipelining Query Execution” Boncz, Zukowski, Nes, CIDR’05

MONETDB/X100: Hyper-Pipelining Query Execution

All vectors together should fit the CPU cache

Optimizer should tune this, given the query characteristics.

“MonetDB/X100: Hyper-Pipelining Query Execution” Boncz, Zukowski, Nes, CIDR’05
Varying the Vector size

Less and less `iterator.next()` and primitive function calls ("interpretation overhead")

MonetDB/X100: Hyper-Pipelining Query Execution
Boncz, Zukowski, Nes,
CIDR’05
Varying the Vector size

Vectors start to exceed the CPU cache, causing additional memory traffic

“MonetDB/X100: Hyper-Pipelining Query Execution” Boncz, Zukowski, Nes, CIDR’05
Varying the Vector size

The benefit of selection vectors

"MonetDB/X100: Hyper-Pipelining Query Execution" Boncz, Zukowski, Nes, CIDR’05
MonetDB/MIL materializes columns

CPU cache

RAM

ColumnBM (buffer manager)

(reid) Disk(s)
Benefits of Vectorized Processing

- **100x less Function Calls**
  - iterator.next(), primitives

- **No Instruction Cache Misses**
  - High locality in the primitives

- **Less Data Cache Misses**
  - Cache-conscious data placement

- **No Tuple Navigation**
  - Primitives are record-oblivious, only see arrays

- **Vectorization allows algorithmic optimization**
  - Move activities out of the loop ("strength reduction")

- **Compiler-friendly function bodies**
  - Loop-pipelining, automatic SIMD
Vectorizing Relational Operators

- Project
- Select
  - Exploit selectivities, test buffer overflow
- Aggregation
  - Ordered, Hashed
- Sort
  - Radix-sort nicely vectorizes
- Join
  - Merge-join + Hash-join

“Balancing Vectorized Query Execution with Bandwidth Optimized Storage” Zukowski, CWI 2008
Column-Oriented Database Systems

Efficient Column Store Compression
Key Ingredients

- Compress relations on a per-column basis
  - Columns compress well
- Decompress small *vectors* of tuples from a column into the CPU cache
  - Minimize main-memory overhead
- Use light-weight, CPU-efficient algorithms
  - Exploit processing power of modern CPUs
Key Ingredients

- Compress relations on a per-column basis
  - Columns compress well
- Decompress small *vectors* of tuples from a column into the CPU cache
  - Minimize main-memory overhead

“Super-Scalar RAM-CPU Cache Compression” Zukowski, Heman, Nes, Boncz, ICDE ’06
Vectorized Decompression

Idea:

decompress a vector only

compression:
- between CPU and RAM
- Instead of disk and RAM (classic)
RAM-Cache Decompression

- Decompress vectors on-demand into the cache
- RAM-Cache boundary only crossed once
- More (compressed) data cached in RAM
- Less bandwidth use
Multi-Core Bandwidth & Compression

Performance Degradation with Concurrent Queries

avg query speed (clock normalized)

number of concurrent queries

Intel® Xeon E5410 (Harpertown) - Q6b Normal
Intel® Xeon E5410 (Harpertown) - Q6b Compressed
Intel® Xeon X5560 (Nehalem) - Q6b Normal
Intel® Xeon X5560 (Nehalem) - Q6b Compressed
CPU Efficient Decompression

- Decoding loop over cache-resident vectors of code words
- Avoid control dependencies within decoding loop
  - no if-then-else constructs in loop body
- Avoid data dependencies between loop iterations
Disk Block Layout

- Forward growing section of arbitrary size **code words**
  (code word size fixed per block)
Disk Block Layout

- Forward growing section of arbitrary size code words (code word size fixed per block)
- Backwards growing exception list

“Super-Scalar RAM-CPU Cache Compression” Zukowski, Heman, Nes, Boncz, ICDE’06
Naïve Decompression Algorithm

- Use reserved value from code word domain (MAXCODE) to mark exception positions

```c
int code[n]; /* temporary machine addressable buffer */

/* blow up next vector of b-bit input code words into machine addressable representation */
UNPACK[b](code, input, n);

for(i=j=0; i<n; i++) {
    if (code[i] < MAXCODE) {
        output[i] = DECODE(code[i]);
    } else {
        output[i] = exception[---j]);
    }
}
```

"Super-Scalar RAM-CPU Cache Compression" Zukowski, Heman, Nes, Boncz, ICDE’06
Deterioriation With Exception%

- 1.2GB/s deteriorates to 0.4GB/s
Deterioriation With Exception%

- Perf Counters: CPU mispredicts if-then-else
Patching

- Maintain a *patch-list* through code word section that links exception positions
Patching

- Maintain a *patch-list* through code word section that links exception positions
- After decoding, *patch* up the exception positions with the correct values

"Super-Scalar RAM-CPU Cache Compression" Zukowski, Heman, Nes, Boncz, ICDE’06
Patched Decompression

/* initialize cur to index of first exception within codes */
int cur = first_exception;
int code[n]; /* temporary machine addressable buffer */

/* blow up next vector of b-bit input code words into machine addressable representation */
UNPACK[b](code, input, n);

/* LOOP1: decode all values */
for(int i=0; i<n; i++) {
    output[i] = DECODE(code[i]);
}

/* LOOP2: patch it up */
for(int i=1; cur < n; i++) {
    output[cur] = exception[-i];
    cur = cur + code[cur];
}
Patched Decompression

/* initialize cur to index of first exception within codes */
int cur = first_exception;
int code[n]; /* temporary machine addressable buffer */

/* blow up next vector of b-bit input code words into machine addressable representation */
UNPACK[b](code, input, n);

/* LOOP1: decode all values */
for(int i=0; i<n; i++) {
    output[i] = DECODE(code[i]);
}

/* LOOP2: patch it up */
for(int i=1; cur < n; i++) {
    output[cur] = exception[-i];
    cur = cur + code[cur];
}
Decompression Bandwidth

Patching can be applied to:

- Frame of Reference (PFOR)
- Delta Compression (PFOR-DELTA)
- Dictionary Compression (PDICT)

Makes these methods much more applicable to noisy data

↘ without additional cost

● Patching makes two passes, but is faster!
Column-Oriented Database Systems

Conclusion
Summary (1/2)

- Columns and Row-Stores: different?
  - No fundamental differences
  - Can current row-stores simulate column-stores now?
    - not efficiently: row-stores need change
  - On disk layout vs execution layout
    - actually independent issues, on-the-fly conversion pays off
    - column favors sequential access, row random
- Mixed Layout schemes
  - Fractured mirrors
  - PAX, Clotho
  - Data morphing
Summary (2/2)

- Crucial Columnar Techniques
  - Storage
    - Lean headers, sparse indices, fast positional access
  - Compression
    - Operating on compressed data
    - Lightweight, vectorized decompression
  - Late vs Early materialization
    - Non-join: LM always wins
    - Naïve/Invisible/Jive/Flash/Radix Join (LM often wins)
  - Execution
    - Vectorized in-cache execution
    - Exploiting SIMD
Future Work

- looking at write/load tradeoffs in column-stores
  - read-only vs batch loads vs trickle updates vs OLTP
Updates (1/3)

- Column-stores are update-in-place averse
  - In-place: I/O for each column
  - + re-compression
  - + multiple sorted replicas
  - + sparse tree indices

Update-in-place is infeasible!
Updates (2/3)

- Column-stores use differential mechanisms instead
  - Differential lists/files or more advanced (e.g. PDTs)
  - Updates buffered in RAM, merged on each query
  - Checkpointing merges differences in bulk sequentially
    - I/O trends favor this anyway
      - trade RAM for converting random into sequential I/O
      - this trade is also needed in Flash (do not write randomly!)
  - How high loads can it sustain?
    - Depends on available RAM for buffering (how long until full)
      - Checkpoint must be done within that time
      - The longer it can run, the less it molests queries
    - Using Flash for buffering differences buys a lot of time
      - Hundreds of GBs of differences per server
Updates (3/3)

- Differential transactions favored by hardware trends
- Snapshot semantics accepted by the user community
  - can always convert to serialized

  “Serializable Isolation For Snapshot Databases”
  Alomari, Cahill, Fekete, Roehm, SIGMOD’ 08

- Row stores could also use differential transactions and be efficient!
  - Implies a departure from ARIES
  - Implies a full rewrite

My conclusion:

* a system that combines row- and columns needs differentially implemented transactions.
* Starting from a pure column-store, this is a limited add-on.
* Starting from a pure row-store, this implies a full rewrite.
Future Work

- looking at write/load tradeoffs in column-stores
  - read-only vs batch loads vs trickle updates vs OLTP
- database design for column-stores
- column-store specific optimizers
  - compression/materialization/join tricks → cost models?
- hybrid column-row systems
  - can row-stores learn new column tricks?
    - Study of the minimal number changes one needs to make to a row store to get the majority of the benefits of a column-store
    - Alternative: add features to column-stores that make them more like row stores
Conclusion

- Columnar techniques provide clear benefits for:
  - Data warehousing, BI
  - Information retrieval, graphs, e-science
- A number of crucial techniques make them effective
  - Without these, existing row systems do not benefit
- Row-Stores and column-stores could be combined
  - Row-stores may adopt some column-store techniques
  - Column-stores add row-store (or PAX) functionality
- Many open issues to do research on!