Column Stores vs. Row-Stores: How Different Are They Really?

Authors: Daniel Abadi, Sammuel Madden, Nabil Hachem

Presented By: Aaron Sarson
Introduction

• Column-Store database systems have emerged in recent years
  • MonetDB
  • C-Store

• It is commonly understood that column-stores offer superior performance on I/O intensive tasks
  • However, literature fails to address if these performance gains can be achieved in row-store DBMS

• RQ1. This work investigates if row-store DBMS can achieve similar gains if the physical architecture emulates that of column-stores

• RQ2. The authors look to discover which features/attributes of column-stores DBMS contributes most to the performance advantage over row-stores
Row-Oriented Execution
Emulating Column-Stores in Row-Oriented DBMS

- Vertical Partitioning
- Index-Only Plans
- Materialized Views

Authors outline three alternative physical designs:
Vertical Partitioning

**Cons of this approach:**
1. Position attribute on every column
2. Row-stores have large headers associated with each tuple

Wasted memory and/or bandwidth

Queries perform joins on the Position attribute when retrieving multiple attributes of a single entity/row
Index-Only Plans

• Base relations are stored in standard row-store format
  • Addition: Unclustured B+ tree index on every column (ALL tables)
• Through this approach only access to indices is required, and not the actual data
  • Reduce I/O → No disk access

• Cons of this approach:
  1. Predicate-less columns, require index to be scanned to extract values
     • This is slower than scanning a heap file
Materialized Views

• “Optimal set of materialized views for every query flight”
  • optimal view contains only the required columns

• Pre-computed dataset
  • Allows access to just the data needed to answer a query

• **Advantages** of this approach:
  • No need to store record-ids (index only) or position (vertical partition)
  • Only stores tuple headers once
Column-Oriented Execution
Compression

- **Column-Oriented Databases** ⇒ **low information entropy**
  - Compression algorithms perform better under this condition
- Data sorted on a particular column is super-compressible
  - Can be **run-length encoded**

"Intuitively, data stored in columns is more compressible than data stored in rows"
Compression

• Produces a larger compression ratio
  • Memory Gains
    • Reducing number of disks
    • Power consumption
  • Performance Gains
    • Reduced I/O time → Smaller reads
    • If query executor can operator on compressed data performance can be improved further

• Compression differences are largest in row vs column-stores when:
  1. Column data is sorted
  2. Repeating values are present (runs)

\[
\text{Compression Ratio} = \frac{\text{Uncompressed}}{\text{Compressed}}
\]
Late Materialization

• Column-stores have **entity** information distributed throughout a disk(s)
• Row-stores have **entity** information group together (single record)

**Problem?**
• Most queries access multiple attributes of an entity (i.e., name, address)
• Many database output standards (i.e. JDBC, ODBC) work at an entity-at-a-time

**Solution?**
• At some point, query plans must combine data from multiple columns into rows representing an entity
  • Depending on when this is done → “Early Materialization” or “Late Materialization”
• Early Materialization:
  • Constructs entity using relevant columns and then applies row-store operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Sam</th>
<th>Jennifer</th>
<th>Lucy</th>
<th>Alex</th>
<th>Luke</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nation</th>
<th>England</th>
<th>Canada</th>
<th>France</th>
<th>Italy</th>
<th>Canada</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>City</th>
<th>London</th>
<th>London</th>
<th>Paris</th>
<th>Venice</th>
<th>Waterloo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

SELECT Name, City FROM Customer WHERE Nation = "Canada"
• Early Materialization:
  • Constructs entity using relevant columns and then applies row-store operators

```
SELECT Name, City FROM Customer WHERE Nation = "Canada"
```

Aaron, **Canada**, Toronto
Sam, **England**, London
Jennifer, **Canada**, London
Lucy, **France**, Paris
Alex, **Italy**, Venice
Luke, **Canada**, Waterloo
• Late Materialization:
  • Operates on columns

<table>
<thead>
<tr>
<th>Aaron</th>
<th>Sam</th>
<th>Jennifer</th>
<th>Lucy</th>
<th>Alex</th>
<th>Luke</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Canada</th>
<th>England</th>
<th>Canada</th>
<th>France</th>
<th>Italy</th>
<th>Canada</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Toronto</th>
<th>London</th>
<th>London</th>
<th>Paris</th>
<th>Venice</th>
<th>Waterloo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

SELECT Name, City FROM Customer WHERE Nation = “Canada”
- Late Materialization:
  - Operates on columns

```
<table>
<thead>
<tr>
<th>Aaron</th>
<th>Sam</th>
<th>Jennifer</th>
<th>Lucy</th>
<th>Alex</th>
<th>Luke</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Canada</th>
<th>England</th>
<th>Canada</th>
<th>France</th>
<th>Italy</th>
<th>Canada</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Toronto</th>
<th>London</th>
<th>London</th>
<th>Paris</th>
<th>Venice</th>
<th>Waterloo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
SELECT Name FROM Customer WHERE Nation = "Canada" AND City = "London"
```
• Late Materialization:
  • Operates on columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Aaron</th>
<th>Sam</th>
<th>Jennifer</th>
<th>Lucy</th>
<th>Alex</th>
<th>Luke</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nation</th>
<th>Canada</th>
<th>England</th>
<th>Canada</th>
<th>France</th>
<th>Italy</th>
<th>Canada</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>City</th>
<th>Toronto</th>
<th>London</th>
<th>London</th>
<th>Paris</th>
<th>Venice</th>
<th>Waterloo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

SELECT Name FROM Customer WHERE Nation = “Canada” AND City = “London”
Late Materialization - Advantages

1. Selection and aggregation operators tend to reduce the number of tuples which need to be constructed
   - Think of the number tuples we needed to construct in early materialization

2. Data compressed using column-oriented compression methods must be decompressed during the tuple construction process
   - Early materialization constructs many tuples at start
   - Late materialization constructs few tuples at end

3. Cache performance improved
   - Cache line is populated with related data (High data locality of column-stores)
A Typical Query Structure

```
SELECT c.nation, s.nation, d.year, 
    sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo, 
    supplier AS s, dwd Date AS d
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;
```

Restrict the set of tuples using selection predicates on 1+ dimension tables

Next, perform aggregation often grouping on other table attributes
Traditional Query Plan:

• Perform joins in order of predicate selectivity

```sql
SELECT c.nation, s.nation, d.year, sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo, supplier AS s, dwd date AS d
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;
```

Assuming `c.region = ASIA` is the most selective

1. Join customer and lineorder
2. Filter lineorder → customers from ASIA remain
3. nation of these customers is added to customer-order
Traditional Query Plan:

1. Join supplier and lineorder
2. Filter lineorder → suppliers from ASIA remain
3. nation of these suppliers is added to customer-order

```
SELECT c.nation, s.nation, d.year, 
sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo, 
supplier AS s, dwdate AS d
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;
```
Traditional Query Plan:

```
SELECT c.nation, s.nation, d.year, sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo,
     supplier AS s, dwdate AS d
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;
```

1. Join dorder and lineorder
2. Filter lineorder → customers who ordered between the years 1992 and 1997 remain
3. Year of these customers ordered is added to customer-order

Results of joins to are finally GROUPed and aggregated (i.e. sum)
Late Materialized Query Plan:

- Predicate is applied on column-store

```sql
SELECT c.nation, s.nation, d.year, 
       sum(lo.revenue) as revenue 
FROM customer AS c, lineorder AS lo, 
     supplier AS s, dwdate AS d 
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;
```

1. Filter `c.region` → customers from ASIA remain
2. `CUSTKEY` of these customers is extracted
3. These `CUSTKEY`s are joined with `CUSTKEY`s from the fact table.
   - Resulting in 2 position lists
     - 1 sorted (fact table) and 1 unsorted (dimension table)
   - Lists indicate which tuples pass the predicate (i.e. `c.region = ASIA`)
4. Extract values from out-of-order positions (i.e. `c.nation`) alongside the values from in-order set of positions for the fact table (i.e. `lo.suppkey`, `lo.orderdate`, and `lo.revenue`)
Late Materialized Query Plan:

- Predicate is applied on column-store

```
SELECT c.nation, s.nation, d.year, 
  sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo, 
  supplier AS s, dwd AS d
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;
```

1. Filter s.region $\rightarrow$ customers from ASIA remain
2. SUPPKEY of these suppliers is extracted
3. These SUPPKEYs are joined with SUPPKEYs from the fact table.
   - Resulting in 2 position lists
     - 1 sorted (fact table) and 1 unsorted (dimension table)
     - Lists indicate which tuples pass the predicate (i.e. s.region = ASIA)
4. Extract values from out-of-order positions (i.e. s.nation) alongside the values from in-order set of positions for the fact table (i.e. lo.custkey, lo.orderdate, and lo.revenue)

Repeat once more for d.year predicate
An Alternative Plan – Invisible Join

- Late materialized join that minimizes out-of-order value extraction
  - **How is this accomplished?**
    - Rewriting joins as predicates on foreign key columns in fact table

**PHASE 01: Constructing Hash Tables**

- Apply each predicate to dimension table → list of keys satisfying predicate
- Construct hash table
An Alternative Plan – Invisible Join

**PHASE 02: Extract Fact Table Records**

- Use hash tables to locate records in fact table that satisfy predicate
- Probe hash table with each value in foreign key column
- Intersect positions lists → records which satisfy ALL predicates

![Diagram showing the process of extracting fact table records using hash tables and bitwise operations.](image)
PHASE 03: Extract Dimension Table Records & Execute Query

- Apply list of satisfying positions to fact tables
  - Identify foreign key references in the appropriate dimension table
  - Extract corresponding values

Note: “If dimension table key is sorted, contiguous list of identifiers starting from 1 […], then the foreign key actually represents the position of desired tuple in dimension table”
Experiments
Motivation: C-Store vs System X - SSBM

Figure 5: Baseline performance of C-Store “CS” and System X “RS”, compared with materialized view cases on the same systems.
Motivation: C-Store vs System X - SSBM

Figure 5: Baseline performance of C-Store “CS” and System X “RS”, compared with materialized view cases on the same systems.
Motivation: C-Store vs System X - SSBM

System X supports advanced performance features:
• Partitioning
• Multi-threading

Figure 5: Baseline performance of C-Store “CS” and System X “RS”, compared with materialized view cases on the same systems.
Column-Store Simulation in a Row-Store

Figure 6: (a) Performance numbers for different variants of the row-store by query flight. Here, T is traditional, T(B) is traditional (bitmap), MV is materialized views, VP is vertical partitioning, and AI is all indexes. (b) Average performance across all queries.

Materialized views perform best
Column-Store Simulation in a Row-Store

Note: These are not attempts to emulate column-stores

Outside of MVs, Traditional and Traditional(bitmap) perform best on average

Figure 6: (a) Performance numbers for different variants of the row-store by query. Here, T is traditional, T(B) is traditional (bitmap), MV is materialized views, VP is vertical partitioning, and AI is all indexes. (b) Average performance across all queries.
Column-Store Simulation in a Row-Store

• Why can’t we outperform traditional methods ($T$ and $T(B)$)?
  • Tuple Overheads
    • Tuple overhead is quite large in fully vertical portioned approach
    • Must maintain rids or primary keys with each column $\rightarrow$ tuple construction
      • Adds significant overhead to read operations

• Vertical partitioning (VP) approach is competitive with row store when few columns are selected
  • However, as the number of columns selected grows
    • Tuple headers waste space and redundant rids yield inferior performance
Column-Store Simulation in a Row-Store

• **Indexing Only (IA)** approach has low per-record overhead, but hash joins with fact table are expensive
  • System X is unable to defer joins until later in the query plan
    • Cannot retain rids from fact table after joining with a dimension table
Column-Store Performance

• Column stores → No tuple overhead + low join costs
  • Tuple headers are stored separately from data
  • Column stores rely on position order not keys or rids

• How does it beat RS (MV) as they have similar I/O and no joins are required from same table.
  • With all else being the same CS’ advantage may result from its optimizations
    • Compression
    • Late materialization
    • Block Iteration
    • Invisible Join

Recall: AVG CS is faster than RS (MV)!
Column Store Performance

Figure 7: (a) Performance numbers for C-Store by query flight with various optimizations removed. The four letter code indicates the C-Store configuration: T=tuple-at-a-time processing, t=block processing; l=invisible join enabled, i=disabled; C=compression enabled, c=disabled; L=late materialization enabled, l=disabled. (b) Average performance numbers for C-Store across all queries.
Column Store Performance

Figure 7: (a) Performance numbers for C-Store by query with various optimizations removed. The four letter code indicates the C-Store configuration: T=tuple-at-a-time processing, t=block processing; l=invisible join enabled, i=disabled; C=compression enabled, c=disabled; L=late materialization enabled, l=disabled. (b) Average performance numbers for C-Store across all queries.
Conclusion

• Authors successfully illustrate attempts to reproduce I/O performance of column-stores in row-stores were rather fruitless
  • High tuple reconstruction costs
  • High per tuple overheads
    • Tuple headers
    • rids or primary keys

• Optimizations of column-stores were thoroughly explored
  • Identifying the key advantages over row-stores in late materialization and compression optimizations

• Proposed a new join technique invisible join
  • Extending late materialization via between-predicate rewriting
References