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

Daniel J. Abadi, Samuel Madden and Nabil Hachem
SIGMOD 2008

Presented by:

Souvik Pal
Subhro Bhattacharyya

Department of Computer Science
Indian Institute of Technology, Bombay
Outline

1 Introduction

2 Column-Stores vs. Row-Stores
   - Row-oriented execution
   - Column-oriented execution

3 Experiments

4 Conclusion
Row store: Data are stored in the disk tuple by tuple
Column store: Data are stored in the disk column by column
Column Stores

- A relational DB shows its data as 2D tables of columns and rows

**Example**

<table>
<thead>
<tr>
<th>EmpId</th>
<th>Lastname</th>
<th>Firstname</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smith</td>
<td>Joe</td>
<td>40000</td>
</tr>
<tr>
<td>2</td>
<td>Jones</td>
<td>Mary</td>
<td>50000</td>
</tr>
<tr>
<td>3</td>
<td>Johnson</td>
<td>Cathy</td>
<td>44000</td>
</tr>
</tbody>
</table>

*Table 1: Column store vs Row Store [2]*
Column Stores

- A relational DB shows its data as 2D tables of columns and rows
- Row Store: serializes all values of a row together

Example

<table>
<thead>
<tr>
<th>EmpId</th>
<th>Lastname</th>
<th>Firstname</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smith</td>
<td>Joe</td>
<td>40000</td>
</tr>
<tr>
<td>2</td>
<td>Jones</td>
<td>Mary</td>
<td>50000</td>
</tr>
<tr>
<td>3</td>
<td>Johnson</td>
<td>Cathy</td>
<td>44000</td>
</tr>
</tbody>
</table>

Table 1: Column store vs Row Store [2]

Row Store

1,Smith,Joe,40000;
2,Jones,Mary,50000;
3,Johnson,Cathy,44000;
Column Stores

A relational DB shows its data as 2D tables of columns and rows

- Row Store: serializes all values of a row together
- Column Store: serializes all values of a column together

Example

<table>
<thead>
<tr>
<th>EmpId</th>
<th>Lastname</th>
<th>Firstname</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smith</td>
<td>Joe</td>
<td>40000</td>
</tr>
<tr>
<td>2</td>
<td>Jones</td>
<td>Mary</td>
<td>50000</td>
</tr>
<tr>
<td>3</td>
<td>Johnson</td>
<td>Cathy</td>
<td>44000</td>
</tr>
</tbody>
</table>

Table 1: Column store vs Row Store [2]

Row Store
1, Smith, Joe, 40000;
2, Jones, Mary, 50000;
3, Johnson, Cathy, 44000;

Column Store
1, 2, 3;
Smith, Jones, Johnson;
Joe, Mary, Cathy;
40000, 50000, 44000;
## Column Stores

<table>
<thead>
<tr>
<th>Row Store</th>
<th>Column Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+) Easy to add/modify a record</td>
<td>(+) Only need to read in relevant data</td>
</tr>
<tr>
<td>(-) Might read in unnecessary data</td>
<td>(-) Tuple writes require multiple accesses</td>
</tr>
</tbody>
</table>

**Table 2: Column store vs Row Store [1]**
Column Stores

<table>
<thead>
<tr>
<th>Row Store</th>
<th>Column Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+) Easy to add/modify a record</td>
<td>(+) Only need to read in relevant data</td>
</tr>
<tr>
<td>(-) Might read in unnecessary data</td>
<td>(-) Tuple writes require multiple accesses</td>
</tr>
</tbody>
</table>

Table 2: Column store vs Row Store [1]

Column stores are suitable for read-mostly, read-intensive, large data repositories

- data warehouses
- decision support applications
- business intelligent applications

For performance comparison, the star schema bench mark is used (SSBM)
Outline

1. Introduction

2. Column- Stores vs. Row- Stores
   - Row-oriented execution
   - Column-oriented execution

3. Experiments

4. Conclusion
Outline

1. Introduction

2. Column-Stores vs. Row-Stores
   - Row-oriented execution
   - Column-oriented execution

3. Experiments

4. Conclusion
Simulating a Column-Store in a Row-Store

Column-store performance from a row-store?

- Vertical Partitioning
- Index-only plans
- Materialized Views
Vertical Partitioning

**Features**

- Full vertical partitioning of each relation.
- 1 physical table for each column.

**Figure 2:** Vertical Partitioning [1].
Vertical Partitioning

**Features**

- Primary key of relation may be long and composite
- Integer valued “position” column for each table.
- Thus each table has 2 columns.
- Joins required on “position” attribute for multi-column fetch.

![Vertical Partitioning Diagram]

**Figure 2:** Vertical Partitioning [1].
Vertical Partitioning

Problems

- “Position” attribute: stored for every column
  - wastes disk space and bandwidth
- large header per tuple
  - more space is wasted
- Joining tables for multi-column fetch
  - Hash Join slow
  - Index Join slower

Figure 2: Vertical Partitioning [1].
Index-only plans

<table>
<thead>
<tr>
<th>Date</th>
<th>Store</th>
<th>Cust</th>
<th>Product</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/01</td>
<td>BOS</td>
<td>Mesa</td>
<td>Table</td>
<td>$20</td>
</tr>
<tr>
<td>01/01</td>
<td>NYC</td>
<td>Lutz</td>
<td>Chair</td>
<td>$15</td>
</tr>
<tr>
<td>01/01</td>
<td>BOS</td>
<td>Mudd</td>
<td>Bed</td>
<td>$90</td>
</tr>
</tbody>
</table>

Figure 3: Index-only plans [1].

Features

- Unclustered B+ tree index on each table column
- Plans never access actual tuples on the disk
- Tuple headers not stored, so overhead is less
Index-only plans

Features

- Indices stored as (record-id, value) pairs.
- All rids stored
- No duplicate values stored

<table>
<thead>
<tr>
<th>Date</th>
<th>Store</th>
<th>Cust</th>
<th>Product</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/01</td>
<td>BOS</td>
<td>Mesa</td>
<td>Table</td>
<td>$20</td>
</tr>
<tr>
<td>01/01</td>
<td>NYC</td>
<td>Lutz</td>
<td>Chair</td>
<td>$15</td>
</tr>
<tr>
<td>01/01</td>
<td>BOS</td>
<td>Mudd</td>
<td>Bed</td>
<td>$90</td>
</tr>
</tbody>
</table>

Figure 3: Index-only plans [1].
Index-only plans

Problems

- Separate indices may require full index scan which is slow
- Solution: Composite indices required to answer queries directly

Example

```
SELECT AVG(SALARY) FROM EMP WHERE AGE>40
```
Materialized views

Features

- Optimal set of MVs created for given query
- Contains only those columns required to answer the query.
- Tuple headers are stored just once per tuple
- Provides just the required amount of data

Problems

- Query should be known in advance
1 Introduction

2 Column-Stores vs. Row-Stores
   - Row-oriented execution
   - Column-oriented execution

3 Experiments

4 Conclusion
Optimizations in Column-Oriented DBs

- Compression
- Late Materialization
- Block Iteration
- Invisible Join
Compression

Features
- Low information entropy in columns than rows
- Decompression performance more valuable than compression achievable

Advantages
- Low disk space
- Lesser I/O
- Performance increases if queries executed directly on compressed data

Figure 4: Compression [1].
Late Materialization

Information about entities stored in different tables.
Most queries access multiple attributes of an entity.

Naive column-store approach - Early Materialization

- Read necessary columns from disk
- Construct tuples from component attributes
- Perform normal row-store operations of these tuples
- Much of performance potential unused
Late Materialization

Features

- Keep data in columns and operate on column data until late into the query plan.
- Intermediate “position” lists need to be created.
- Required for matching up operations performed on different columns.

Example

```
SELECT R.a FROM R WHERE R.c = 5 AND R.b = 10
```

- Output of each predicate is a bit string
- Perform Bitwise AND
- Use final position list to extract R.a
Late Materialization

Advantages

- Selection and Aggregation limits the number of tuples generated
- Compressed data need not be decompressed for creating tuples
- Better cache performance – PAX
- Block iteration works better on columns than on rows
Partition Attributes Across (PAX)

**Features**
- column interleaving
- minimal row reconstruction cost
- only relevant data in cache
- minimizes cache misses
- effective when applying querying on a particular attribute

**Figure 5:** PAX [3].
Block Iteration

Features

- Operators operate on blocks of tuples at once
  - Iterate over blocks of tuples rather than a single tuple
  - Avoids multiple function calls on each tuple to extract data
  - Data is extracted from a batch of tuples
- Fixed length columns can be operated as arrays
  - Minimizes per-tuple overhead
  - Exploits potential for parallelism
Figure 6: Star Schema Benchmark [4].
Invisible Join

Example

```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 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;
```

- Find total revenue from customers who live in ASIA
- and who purchase from an Asian supplier between 1992 and 1997
- grouped by nation of customer, nation of supplier and year of transaction
Invisible Join

Traditional Plan

Pipelines join in order of predicate selectivity.
Disadvantage: misses out on late materialization

Late materialized join:
Disadvantage

After join the list of positions for dimension tables are unordered

Group by columns in dimension tables need to be extracted in out-of-position order.

Figure 7: Late materialization [1].
Invisible Join

Phase 1

Figure 8: Phase 1 [4].
Invisible Join

Phase 2

Figure 9: Phase 2 [4].
Figure 10: Phase 3 [4].
Invisible Join

Between-Predicate Rewriting

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, 3

Range [1-3] (between-predicate rewriting)

Hash Table Containing Keys 01011997, 01021997, and 01031997

Figure 11: Between-Predicate Rewriting [1].
Outline

1. Introduction

2. Column-Stores vs. Row-Stores
   - Row-oriented execution
   - Column-oriented execution

3. Experiments

4. Conclusion
Goal

- Performance comparison of C-Store with R-Store
- Performance comparison of C-Store with column-store simulation on a R-Store
- Finding the best optimization for a column-store
- Comparison between invisible join and denormalized table
Figure 12: C-Store (CS) and System-X (RS) [4].
C-Store(CS) vs. System-X(RS)

- First three rows as per expectation.
- For CS(Row-MV) materialized data is stored as strings in C-store.
- Expected that both RS(MV) and CS(Row-MV) will perform similarly
- However RS(MV) performs better
  - No support for multi-threading and partitioning in C-Store.
  - Disabling partitioning in RS(MV) halves performance
  - Difficult to compare across systems
- C-Store(CS) 6 times faster than CS(Row-MV)
  - Both read minimal amount of data from disk to answer a query
  - I/O savings- not the only reason for performance advantage
Column store simulation in Row store

- Traditional
- Vertical Partitioning: Each column is a relation
- Index-only plans: B+Tree on each column
- Materialized Views: Optimal set of views for every query
Column store simulation in Row store

- MV < T < VP < AI (time taken)
- Without partitioning, T ≈ VP
- Vertical partitioning: Tuple Overhead

<table>
<thead>
<tr>
<th></th>
<th>1 Column</th>
<th>Whole Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>4 GB</td>
<td></td>
</tr>
<tr>
<td>VP</td>
<td>1.1 GB</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>240 MB</td>
<td>2.3 GB</td>
</tr>
</tbody>
</table>

- Index-only plans: Column Joins
  - Hash Join: takes a long time
  - Index Join: high index access overhead
  - Merge Join: unable to skip sort step

Figure 13: Column store simulation in Row store [4].
Breakdown of Column-Store Advantages

- Start with C-Store
- Remove optimizations one by one
- Finally emulate Row-Store
- Late materialization improves 3 times
- Compression improves 2 times
- Invisible Join improves 50%
- Block processing improves 5-50%

![Graph showing performance comparisons between different configurations.](image)
Outline

1. Introduction

2. Column-Stores vs. Row-Stores
   - Row-oriented execution
   - Column-oriented execution

3. Experiments

4. Conclusion
C-Store emulation on R-Store is done by vertical partitioning, index plans

Emulation does not yield good performance

Reasons for low performance by emulation
- High tuple reconstruction costs
- High tuple overhead

Reasons for high performance of C-Store
- Late Materialization
- Compression
- Invisible Join

