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

Daniel J. Abadi (Yale)
Samuel R. Madden (MIT)
Nabil Hachem (AvantGarde)

Presented By Pragnya Addala
Introduction

- Significant amount of excitement and recent work on column-oriented database systems – “column-stores”

- On analytical workloads, these are found to perform order of magnitude better than traditional row-oriented database systems – “row stores”

- Elevator pitch: “column-stores are more I/O efficient for read-only queries since they only have to read from disk (or from memory) those attributes accessed by a query”
Motivation

- **Common assumption.**

“One can obtain the performance benefits of a column-store using a row-store; either by vertically partitioning the schema, or by indexing every column so that the columns can be accessed independently.”

**Is this assumption valid?**

- Can we adapt our row-store to get column-store performance?

- If not, what makes column-store not simulatable?
• In row store, data is stored in the disk tuple by tuple.

• Where as in column store, data is stored in the disk column by column
Background

• Most of the queries do not process all the attributes of a particular relation.

• For example the query
  
  Select c.name and c.address
  From CUSTOMES as c
  Where c.region=Mumbai;

• Only processes three attributes of the relation CUSTOMER. But the customer relation can have more than three attributes.

• Column-stores are more I/O efficient for read-only queries as they read, only those attributes which are accessed by a query.
Background

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

<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>

Introduction
Motivation
Background
Paper Methodology
SSBM
Row-Oriented Execution
- Vertical Partitioning
- Index-only Plans
- Materialized Views
Column-Oriented Execution
- Compression
- Late Materialization
- Block Iteration
- Invisible Join
Experiments
Conclusion
Take Away Message
Q&A
Background

• **Why column stores?**

  - Can be significantly faster than row stores for some applications
    - Fetch only required columns for a query
    - Better cache effects
    - Better compression (similar attribute values within a column)

• But can be slower for other applications
  - OLTP with many row inserts, ..

• Long war between the column store and row store camps
  - This paper tries to give a balanced picture of advantages and disadvantages, after adding/subtracting a number of optimizations for each approach
Paper Methodology

- Comparing row-store vs. column-store is dangerous/borderline meaningless

- Instead, compare row-store vs. row-store and column-store vs. column-store
  - Simulate a column-store inside of a row-store
  - Remove column-oriented features from column-store until it behaves like a row-store
SSBM is a data warehousing benchmark derived from TPC-H

It consists of a single fact table LINE-ORDER

There are four dimension table.
  - CUSTOMER
  - PART
  - SUPPLIER
  - DATE

LINEORDER table consists of 60,000,000 tuples

SSBM consists of thirteen queries divided into four categories
Star Schema Benchmark (SSBM)

Figure 1: Schema of the SSBM Benchmark
Row-Oriented Execution

- Now the simplistic view about the difference in storage layout leads to the assumption that one can obtain the performance benefits of a column-store using a row-store by making some changes to the physical structure of the row store.

- These changes can be
  - Vertically partitioning
  - Using index-only plans
  - Using materialized views
Vertical Partitioning

• Process:
  – Full Vertical partitioning of each relation
    • Each column = 1 Physical table
    • This can be achieved by adding integer position column to every table
    • Adding integer position is better than adding primary key
  – Join on Position for multi column fetch

• Problems:
  – “Position” - Space and disk bandwidth
  – Header for every tuple – further space wastage
    • e.g. 24 byte overhead in PostgreSQL
Vertical Partitioning: Example

- Each attribute is a two-column table: (values, position)
Index-only Plans

- **Process:**
  - Add B+Tree index for every Table.column
  - Plans never access the actual tuples on disk
  - Headers are not stored, so per tuple overhead is less

- **Problem:**
  - Separate indices may require full index scan, which is slower
  - Eg: SELECT AVG(salary)
    FROM emp
    WHERE age > 40
  - Composite index with (age, salary) key helps.
  - Slow Tuple Construction
Index-only Plans: Example

- Unclustered B+Tree index for every column of every table
Materialized Views

- **Process:**
  - Create ‘optimal' set of MVs for given query workload
  - **Objective:**
    - Provide just the required data
    - Avoid overheads
    - Performs better

- **Expected to perform better than other two approaches**

- **Problems:**
  - Practical only in limited situation
  - Requires knowledge of query workloads in advance
Materialized Views: Example

- Select F.custID from Facts as F where F.price > 20
Column-Oriented Execution

- Different optimizations for column oriented database
  - Compression
  - Late Materialization
  - Block Iteration
  - Invisible Join
Compression

- Low information entropy (high data value locality) leads to high compression ratio

- **Advantages**
  - Disk Space is saved
  - Less I/O
  - CPU cost decrease if we can perform operation without decompressing

- Light weight compression schemes do better
Compression

- If data is sorted on one column that column will be super-compressible in row store
- eg. Run-length encoding
Late Materialization

- Most query results entity-at-a-time not column-at-a-time
- So at some point of time, multiple column must be combined
- One simple approach is to join the columns relevant for a particular query
- But further performance can be improve using late-materialization
Late Materialization

- Delay Tuple Construction
- Might avoid constructing it altogether
- Intermediate position lists might need to be constructed
- Eg: 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

- **Advantages**
  - Unnecessary construction of tuple is avoided
  - Direct operation on compressed data
  - Cache performance is improved (PAX)
Block Iteration

- Operators operate on blocks of tuples at once
- Iterate over blocks rather than tuples
- Like batch processing
- If column is fixed width, it can be operated as an array

**Advantages:**
- Minimizes per-tuple overhead
- Exploits potential for parallelism

- Can be applied even in Row stores – IBM DB2 implements it
Invisible Join

• Queries over data warehouse (particularly modeled with star schema) often have following structure
  – Restrict set of tuple in the fact table using selection predicates on dimension table
  – Perform some aggregation on the restricted fact table
  – Often grouping by other dimension table attribute

• For each selection predicate and for each aggregate grouping, join between fact table and dimension table is required
Invisible Join

```
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;
```

- Find Total revenue from Asian customers who purchase a product supplied by an Asian supplier between 1992 and 1997 grouped by nation of the customer, supplier and year of transaction
Invisible Join

- Traditional plan for this type of query is to pipeline join in order of predicate selectivity

- Alternate plan is late materialized join technique

- But both have **disadvantages:**
  - Traditional plan lacks all the advantages described previously of late materialization
  - In the late materialized join technique group by columns need to be extracted in out-of-position order
Invisible Join

- Invisible join is a late materialized join but minimizes the values that need to be extracted out of order

- Invisible join
  - Rewrite joins into predicates on the foreign key columns in the fact table
  - These predicates evaluated either by hash-lookup
  - Or by between-predicate rewriting
Invisible Join

```
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;
```

- Find Total revenue from Asian customers who purchase a product supplied by an Asian supplier between 1992 and 1997 grouped by nation of the customer, supplier and year of transaction
Invisible Join – Phase 1

Each predicate is applied to the appropriate dimension table to extract a list of dimension table keys that satisfy the predicate.
Each hash table is used to extract the positions of tuples in the fact table that satisfy the corresponding predicate.
The third phase uses the list of satisfying positions \( P \) in the fact table to get foreign key values and hence needed data values from the corresponding dimension table.
Invisible Join

- **Between-Predicate rewriting**
  - Use of range predicates instead of hash lookup in phase 1
  - Useful if contiguous set of keys are valid after applying a predicate
  - Dictionary encoding for key reassignment if not contiguous
  - Query optimizer is not altered. Predicate is rewritten at runtime
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>
Experiments

• Goal
  – Comparison of attempts to emulate a column store in a row-store with baseline performance of C-Store
  – Is it possible for an unmodified row-store to obtain the benefits of column oriented design?
  – Effect of different optimization techniques in column-store
Experimental Setup

• Environment
  – 2.8GHz Dual Core Pentium(R) workstation
  – 3 GB RAM
  – RHEL 5
  – 4 disk array mapped as a single logical volume
  – Reported numbers are average of several runs
  – Warm buffer (30% improvement for both systems)
    • Data read exceeds the size of buffer pool
C-Store vs. Commercial Row-Oriented DB

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>RS</th>
<th>RS (MV)</th>
<th>CS</th>
<th>CS (Row-MV)</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>2.7</td>
<td>1.0</td>
<td>0.4</td>
<td>16.0</td>
<td>25.7</td>
</tr>
<tr>
<td>1.2</td>
<td>2.0</td>
<td>1.0</td>
<td>0.1</td>
<td>9.1</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>1.5</td>
<td>0.2</td>
<td>0.1</td>
<td>8.4</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>43.8</td>
<td>15.5</td>
<td>5.7</td>
<td>33.5</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>44.1</td>
<td>13.5</td>
<td>4.2</td>
<td>23.5</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>46.0</td>
<td>11.8</td>
<td>3.9</td>
<td>22.3</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>43.0</td>
<td>16.1</td>
<td>11.0</td>
<td>48.5</td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>42.8</td>
<td>6.9</td>
<td>4.4</td>
<td>21.5</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>31.2</td>
<td>6.4</td>
<td>7.6</td>
<td>17.6</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>6.5</td>
<td>3.0</td>
<td>0.6</td>
<td>17.4</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>44.0</td>
<td>29.2</td>
<td>8.2</td>
<td>48.6</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>14.1</td>
<td>22.4</td>
<td>3.7</td>
<td>38.4</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>12.2</td>
<td>6.4</td>
<td>2.6</td>
<td>32.1</td>
<td></td>
</tr>
</tbody>
</table>

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

**RS**: Base System X
**RS (MV)**: System X with optimal collection of MVs
**CS**: Base C-Store case
**CS (Row-MV)**: Column store constructed from RS(MV)
**System X**: Commercial row-oriented database
Results and Analysis

• From the graph we can see
  – C-Store out performs System X by a
    • Factor of six in the base case
    • Factor of three when System X use materialized view

• However CS (Row-MV) perform worse than RS (MV)
  – System X provide advance performance feature
  – C-Store has multiple known performance bottleneck
    • C-Store doesn't support partitioning, multithreading
Column Store simulation in Row Store

- **Partitioning** improves the performance of row store if done on a predicate of the query

- Authors found that it improves the speed by a factor of two

- System X implement star join

- Optimizer will bloom filters if it feels necessary

- Other configuration parameters
  - 32 KB disk pages
  - 1.5 GB maximum memory for sort joins, intermediate result
  - 500 MB buffer pool
Different configuration of System X

- Experimented with five different configurations:
  1. Traditional row oriented representation with bitmap and bloom filter
  2. Traditional (bitmap): Biased to use bitmaps; might be inferior sometimes
  3. Vertical Partitioning: Each column is a relation
  4. Index-Only: B+Tree on each column
  5. Materialized Views: Optimal set of views for every query
T – Traditional, T(B) – Traditional(bitmap), MV – materialized views, VP – vertical partitioning, AI – All indexes
Different configuration of System X

- Better performance of traditional system is because of partitioning.
- Partitioning on orderdate

T – Traditional,
T(B) – Traditional(bitmap),
MV – materialized views,
VP – vertical partitioning,
AI – All indexes
Different configuration of System X

- **Materialized view** performs best
- **Index only** plans are the worst
- Expensive column joins on fact table
  - System X use hash join by default
  - Nested loop join, merge join also does not help
Column Store simulation in Row Store: Analysis

Vertical Partitioning

- **Tuple overheads:**
  - LineOrder Table
    - 60 million tuples, 17 columns
    - 8 bytes of overhead per row
    - 4 bytes of record-id

<table>
<thead>
<tr>
<th>Tuple Header</th>
<th>TID</th>
<th>Col. Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1 column</th>
<th>Whole table</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS</td>
<td>0.7-1.1 GB</td>
<td>4 GB</td>
</tr>
<tr>
<td>CS</td>
<td>240 MB</td>
<td>2.3 GB</td>
</tr>
</tbody>
</table>

For SSBM scale 10 `lineorder` table
Column Store simulation in Row Store: Analysis

All Indexes

- Pretty much all queries require a column to be extracted (in the SELECT clause) that has not yet been accessed, e.g.:
  
  ```
  SELECT store_name, SUM(revenue) 
  FROM Facts, Stores 
  WHERE fact.store_id = stores.store_id 
  AND stores.area = "NEW ENGLAND" 
  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
Column Store simulation in Row Store: Analysis

- 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, ..
Column Store Performance

- Column Store perform better than the best case of row store (4.0sec Vs 10.2sec)
- Though they access the same amount of I/O is similar
# Tuple overhead and Join costs

<table>
<thead>
<tr>
<th>Row Store</th>
<th>Column Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store the record-id explicitly</td>
<td>Don’t explicitly Store the record-id</td>
</tr>
<tr>
<td>Headers are stored with each column</td>
<td>Header are stored in separate column</td>
</tr>
<tr>
<td>Use index-based merge join</td>
<td>Use merge join</td>
</tr>
</tbody>
</table>

- These differences are not fundamental
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;
Different configuration of System X

\[ T = \text{tuple-at-a-time processing}; \]
\[ t = \text{block processing}; \]
\[ I = \text{invisible join enabled}; \]
\[ i = \text{disabled}; \]
\[ C = \text{compression enabled}; \]
\[ c = \text{disabled}; \]
\[ L = \text{late materialization enabled}; \]
\[ l = \text{disabled}; \]
Breakdown of Column-Store Advantages

• **Block processing** improves the performance by a factor of 5% to 50%

• **Compression** improves the performance by almost a factor of two on avg

• **Late materialization** improves performance by almost a factor of three

• **Invisible join** improves the performance by 50-75%
In Conclusion..

• Common assumption..

“One can obtain the performance benefits of a column-store using a row-store; either by vertically partitioning the schema, or by indexing every column so that the columns can be accessed independently.”

is false!!

• Can we adapt our row-store to get column-store performance?
  – Currently, NO.

• If not, what makes column-store not simulatable?
  – Optimizations at query execution level
Conclusion

- To simulate column store in row store, techniques like
  - Vertical partitioning
  - Index only plan
do not yield good performance

- High per-tuple overheads, high tuple reconstruction costs are the reason

- Where as in column store
  - Late materialization
  - Compression
  - Block iteration
  - Invisible join
are the reasons for good performance
Take Away Message

- Column-store simulation performs poorly on today’s row store systems.

- Successful column store simulation in row store will require:
  - Virtual record-ids
  - Reduced tuple over head
  - Fast merge join
  - Run length encoding across multiple tuples
  - Operating directly on compressed data
  - Block processing
  - Invisible join
  - Late materialization...
Q&A

• In the future, where do you see this going? Column store simulation in row store or row store simulation in column store? Which is be more widely used?

• How do you think read/write locks for column store work?

• Assert for insert etc.. ?

• Secondary indexes?

• When do you use row, and when column? – (SAP HANA)

• Any other general comments/questions?