Column-Store: An Overview

Row-Store (Classic DBMS)
- Store one tuple at-a-time

Column-Store
- Store one column at-a-time

Row-Store vs Column-Store

Tuple Insertion:
- Row-Store: Fast
- Column-Store: Requires multiple seeks

Queries:
- Row-Store: May access more data than needed
- Column-Store: Read only relevant data

Perfect for analytics!

Column-Store Optimizations

Operating on columns enables and/or is combined with the following optimizations:

- **Compression**
  Compress values per column

- **Late Tuple Materialization**
  Construct tuples as late as possible

- **Block Iteration**
  Pass blocks of values between operators

- **Invisible Join**
Compression

Compress column data

Why Compression

Advantages of compression in general:

- Lower storage space requirements
  Minor
- Better I/O performance
  Read fewer data (from disk, SSD, or RAM), gain from cache locality
- Better query processing performance
  Typically when operating directly on compressed data

Why Column Store

Compress column data:

- One column at a time
  Data in a column more similar than data across columns

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
<th>City</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred Flintstone</td>
<td>858-123-4567</td>
<td>San Diego</td>
<td>CA</td>
</tr>
<tr>
<td>Barney Rubble</td>
<td>619-000-0000</td>
<td>San Diego</td>
<td>CA</td>
</tr>
<tr>
<td>Maggie Simpson</td>
<td>415-999-2222</td>
<td>San Francisco</td>
<td>CA</td>
</tr>
<tr>
<td>James Bond</td>
<td>212-007-0000</td>
<td>New York</td>
<td>NY</td>
</tr>
</tbody>
</table>
Compression Example

- **Run-length encoding**
  Replace list of identical values by pair (value, count)

<table>
<thead>
<tr>
<th>State</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>CA x3</td>
</tr>
<tr>
<td>CA</td>
<td>CA</td>
</tr>
<tr>
<td>CA</td>
<td>NY x1</td>
</tr>
<tr>
<td>NY</td>
<td>NY</td>
</tr>
</tbody>
</table>

Good for sorted columns

<table>
<thead>
<tr>
<th>State</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>CA</td>
</tr>
<tr>
<td>NY</td>
<td>NY</td>
</tr>
<tr>
<td>Non-sorted</td>
<td>Sorted</td>
</tr>
<tr>
<td>CA</td>
<td>CA</td>
</tr>
<tr>
<td>NY</td>
<td>NY</td>
</tr>
</tbody>
</table>

Enables query processing on compressed data directly

- e.g., Select persons in CA

<table>
<thead>
<tr>
<th>State</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA x3</td>
<td>CA</td>
</tr>
<tr>
<td>NY x1</td>
<td>NY</td>
</tr>
<tr>
<td>uncompress</td>
<td>St = &quot;CA&quot;</td>
</tr>
<tr>
<td>CA</td>
<td>CA</td>
</tr>
<tr>
<td>CA</td>
<td>CA</td>
</tr>
<tr>
<td>NY</td>
<td>NY</td>
</tr>
</tbody>
</table>
Compression Example

- Run-length encoding
  Replace list of identical values by pair (value, count)
  Enables query processing on compressed data directly

  e.g., Select persons in CA

```
<table>
<thead>
<tr>
<th>State</th>
<th>CA</th>
<th>NY</th>
</tr>
</thead>
<tbody>
<tr>
<td>St = “CA”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>‘CA’</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>‘NY’</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

Other Compression Algos

- Dictionary Encoding
  Replace frequent patterns with smaller fixed length codes:
  e.g., instead of string values "Dasgupta" → 0, "Freund" → 1,
  "Papakonstantinou" → 2
  Commonly used in row-stores also, since it enables fixed length fields, therefore random access.

- Bit-Vector Encoding
  Create for each possible value a bit vector with 1s in the positions containing the value: Useful for small domains.
  (Covered in the indexing section.)

- Heavyweight, Variable-Length Compression Schemes
  e.g., Huffman: Excellent compression ratio but (1) no random access (2) possibly poor decompression CPU performance
  Currently not used – they are good for selected workloads

Late Tuple Materialization

Create tuples as late in the query plan as possible
Early Tuple Materialization

Naïve method of query evaluation in a column store:
- Read relevant columns & create tuples
- Run query at a tuple-level as usual

\[\xrightarrow{\text{Query Processing: Rows}}\]

Storage: Columns

\[\xrightarrow{\text{Create tuples}}\]

\[
\begin{array}{c|c|c|c}
\text{Name} & \text{Phone} & \text{City} & \text{State} \\
\hline
\text{Flintstone} & 858 & SD & CA \\
\text{Barney} & 619 & SD & CA \\
\text{Simpson} & 415 & SF & CA \\
\text{Bond} & 212 & NY & NY \\
\end{array}
\]

\[\xrightarrow{\text{σ City = "SD" ∧ State = "CA"}}\]

e.g., \(\Pi \text{Name} \sigma \text{City = "SD" ∧ State = "CA"} \)

\[\xrightarrow{\text{Create tuples}}\]

\[
\begin{array}{c|c|c|c}
\text{Name} & \text{City} & \text{State} \\
\hline
\text{Flintstone} & SD & CA \\
\text{Barney} & SD & CA \\
\text{Simpson} & SF & CA \\
\text{Bond} & NY & NY \\
\end{array}
\]
Early Tuple Materialization

Example: SELECT Name FROM Person WHERE City="SD" AND State="CA"

- Create tuples
- σ$_{City = "SD" \land State="CA"}$

Selects tuples with City equal to "SD" and State equal to "CA".

Diagram:

- Tuples for Flintstone and Barney.
- Tuples for Flintstone and Barney are selected.
- Reads relevant columns only.
Late Tuple Materialization

• Create tuples as late in the plan as possible

Column-Store Optimizations > Late Tuple Mat.

Storage & Query Processing: Columns

---

e.g., SELECT Name FROM Person WHERE City="SD" AND State="CA"

---

Position lists
Late Tuple Materialization

Example: SELECT Name FROM Person WHERE City="SD" AND State="CA"

- Extract
- City = "SD"
- State = "CA"

Position lists can be represented as:

- Bit Vectors
- Ranges of positions

Run-length Encoding Revisited
Late Materialization Benefits

- Avoid materializing certain tuples since they may be filtered out before being materialized (Reminds of pushing selections down.)
- Avoid data decompression which has to be done when a tuple is materialized
- Leverage improved cache locality which exists when operating on a single column
- Leverage optimizations for fixed-width attributes which would not be possible if operating on the tuple level, since a tuple with at least one variable-width attribute becomes variable-width

Block Iteration

- Row Store
  - Pass single tuples between operators
  - Extract attribute value through function calls

- Column Store
  - Pass blocks of values between operators
  - No need for attribute extraction
Invisible Join

Join Optimization for Star Schemas

Late Tuple Materialization requires out of order access when we join tables

\[ \text{e.g.,} \]

\begin{align*}
\text{R} & : 1 & 1 & 2 & 3 & 4 & 5 \\
\text{S} & : 1 & 2 & 3 & 4 & 2 & 5
\end{align*}

out of order access to S

Optimization for joins on Star Schemas
- Make sure that the fact table is accessed in order
- Reduce the amount of data that are accessed out of order from the dimension tables
- Reminiscent of bitmap intersection technique
Invisible Join Example

**Column Store Optimizations**

**Schema**

- **LineOrder**
  - order_key
  - cust_key
  - supp_key
  - date
  - revenue

- **Customer**
  - cust_key
  - region
  - nation

- **Supplier**
  - supp_key
  - region
  - nation

**Date**
- date
- year

---

**Query**

```
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;
```

---

**Data**

<table>
<thead>
<tr>
<th>LineOrder</th>
<th>Supplier</th>
<th>Customer</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>order_key</td>
<td>cust_key</td>
<td>region</td>
<td>year</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>010197</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>010297</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>010297</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>23233</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>2</td>
<td>010397</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>010497</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>2</td>
<td>010597</td>
</tr>
</tbody>
</table>

---

Example taken from VLDB 09 Tutorial by Harizopoulos, Abadi, Böcz.
Invisible Join Example

**Step 1: Apply selections on dimension tables**

- **Supplier**
  - `σ region = "Asia"` → Hashable {1, 3}

- **Customer**
  - `σ region = "Asia"` → Hashable {1, 2, 3}

- **Date**

**Step 2: Find fact table tuples satisfying all selections on dimension tables**

- `Hashtable {1, 3}

- `Hashtable {1, 2, 3}

- `Hashtable {010197, 010297, 010397}`
Invisible Join Example

Column Store Optimizations

Step 2: Find fact table tuples satisfying all selections on dimension tables

*Diagram showing a join operation with dimension tables.*

Invisible Join Example

Column Store Optimizations

Step 3: Extract data from dimension tables

*Tables showing sales data from customer, supplier, and lineorder tables.*

---

Invisible Join Example

Column Store Optimizations

Step 3: Extract data from dimension tables

*Tables showing additional data from customer, supplier, and lineorder tables.*
Column Store Optimizations

Invisible Join Example

Step 3: Extract data from dimension tables

<table>
<thead>
<tr>
<th>Lineorder Date</th>
<th>Date Date Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>010197</td>
<td>010197 1997</td>
</tr>
<tr>
<td>010197</td>
<td>1997</td>
</tr>
<tr>
<td>010197</td>
<td>010197</td>
</tr>
<tr>
<td>010297</td>
<td>1997</td>
</tr>
<tr>
<td>010297</td>
<td>010297</td>
</tr>
<tr>
<td>010297</td>
<td>1997</td>
</tr>
<tr>
<td>010397</td>
<td>010397</td>
</tr>
<tr>
<td>010397</td>
<td>1997</td>
</tr>
</tbody>
</table>

Optimizations Summary

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

Comparing Optimizations

What is the speedup of each column-store optimization?

- Compression: $2\times$ (avg) / $10x$ (sorted)
- Late Tuple Materialization: $3x$
- Block Iteration: $1.05-1.5x$
- Invisible Join: $1.5-1.75x$

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

- How better is a column-store than a row-store? 
  Heated debate: (Exaggerated) claims of performance up to 16,200x
- Can we simulate it in a row-store and get the performance benefits or does the row-store have to be internally modified? 
  Another heated debate: Many papers on the topic
- Can we create a hybrid that will accommodate both transactional and analytics workloads?

Column-Store Simulation

A column-store can be simulated in a row-store through:

- Vertical Partitioning
  Create one table per column
- Index-only Plans
  Create one index per column & use only indexes
- Materialized Views
  Create views of interest for given workload
- C-Table

Column-Store Simulation

**Vertical Partitioning**

- Create one table per column

ID: Val 

Used to link these tables (column-stores use the order)
Vertical Partitioning

- Column-Store Simulation

**Overhead from storing tuple-ID**

**Excessive joins**

**Index-only Plans**
- Create one index per column & use only indexes
- Data are kept as rows but are never accessed
Column-Store Simulation
Index-only Plans

e.g., SELECT Name FROM Person WHERE City="SD" AND State="CA"

\[
\pi_{\text{Name}} \left( \sigma_{\text{City} = "SD" \land \text{State} = "CA"}(\text{Scan(\text{INDEX}))}) \right)
\]
Index-only Plans

\[ \pi_{\text{Name}} (\sigma_{\text{City}=\text{"SD"}} \land \sigma_{\text{State}=\text{"CA"}} \text{Scan INDEX}) \]

- Avoids table access
- Have to scan index for attributes without predicate
- Create indexes with composite keys!

Materialized Views

- Create optimal view(s) for each query
  For each relation involved in the query, create a view including only columns needed to answer the query
  \[ Q = \pi_{a,b} (\sigma_{b='x'} (R)) \]

- Requires workload knowledge

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
<th>City</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flintstone</td>
<td>555-1234</td>
<td>SD</td>
<td>CA</td>
</tr>
<tr>
<td>Barney</td>
<td>666-5432</td>
<td>SD</td>
<td>CA</td>
</tr>
<tr>
<td>Simpson</td>
<td>777-0987</td>
<td>SF</td>
<td>CA</td>
</tr>
<tr>
<td>Bond</td>
<td>888-4321</td>
<td>NY</td>
<td>NY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>City</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flintstone</td>
<td>SD</td>
<td>CA</td>
</tr>
<tr>
<td>Barney</td>
<td>SD</td>
<td>CA</td>
</tr>
<tr>
<td>Simpson</td>
<td>SF</td>
<td>CA</td>
</tr>
<tr>
<td>Bond</td>
<td>NY</td>
<td>NY</td>
</tr>
</tbody>
</table>
Column-Store Simulation

**C-Table**

- Extend vertical partitioning with run-length encoding
- Rewrite queries to operate on the compressed data

<table>
<thead>
<tr>
<th>Start Pos</th>
<th>Value</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pretty close to performance of native column-store

**Back to our question:**

**Column-Store vs Row-Store**

- Column-Stores have a definite advantage on analytic workflows…
- …but Row-Stores can be improved by taking lessons from them
Implementations

Open Source

C-Store

Commercial

SYBASE
INFOBRIGHT
Vertica
SQL Server 2012
PARACCEL