Row-Store / Column-Store / Hybrid-Store

Kevin Sterjo
Technische Universität München
Department of Informatics
Chair for Database Systems
Garching, 12. December 2017
Table of contents

1. Introduction
2. Row Store
3. Column Store
4. Hybrid Store
5. Implementation
6. Questions
Table of contents

1. Introduction
2. Row Store
3. Column Store
4. Hybrid Store
5. Implementation
6. Questions
Introduction

Disk Resident Database Systems (DRDB)
- Data is stored on disk
- May be cached into memory for access

Main Memory Database Systems (MMDB)
- Data is stored permanently on main physical memory
- Backup on disk (if needed)
Introduction

Why use MMDB?

- Lower I/O cost
- Access time
- Directly accessible by the processor(s)
- Getting cheaper
Table of contents

1. Introduction
2. Row Store
3. Column Store
4. Hybrid Store
5. Implementation
6. Questions
Row Store

Traditional way data is physically stored
All attributes of each tuple stored subsequently

Figure 1: Logical Row Store. Source: own representation based on [1]
Row Store

Whole row written in a single operation
More preferable for OLTP-oriented databases
What happens when we need to access only one attribute?

Figure 2: Logical Row Store with query. Source: own representation based on [1]
Row Store

Attributes of different types
Compression algorithms difficult to implement compared to other layouts
Use of dictionaries
Huffman encoding
Table of contents

1. Introduction
2. Row Store
3. **Column Store**
4. Hybrid Store
5. Implementation
6. Questions
Column Store

Dates back to the ’70s
Attributes depicted by columns stored contiguously

![Logical Column Store Diagram](source)

Figure 3: Logical Column Store. Source: own representation based on [1]
Column Store

Attributes of the same type
Perform an order of magnitude better on analytical workloads
Successfully implemented in OLAP-oriented databases
Compression also a factor
But…
Write operations and tuple construction problematic

Figure 4: Memory alignment and resulting access patterns for row and column store.
Source: own representation based on [2]
Table of contents

1. Introduction
2. Row Store
3. Column Store
4. Hybrid Store
5. Implementation
6. Questions
Hybrid Store

Indecisiveness between row and column store
What happens when advantages of them are required?
Combination of both techniques
Insert and update intense data stored in row store component
Data used for analytical processes stored in column store
Hybrid Store

Transactional processing carried out on a dedicated OLTP database system
Additional Data Warehouse implemented for business intelligence query processing
ETL (Extract-Transform-Load) in specific intervals
Data staleness
Table of contents

1. Introduction
2. Row Store
3. Column Store
4. Hybrid Store
5. Implementation
6. Questions
Table of contents

1. Introduction
2. Row Store
3. Column Store
4. Hybrid Store
5. Implementation
   1. Initial Effort
   2. Evaluation
   3. Approach
   4. Results
6. Questions
Implementation
Initial Effort

Use of basic C++ objects, namely “vector” and “struct”
Table used provided as CSV file consisting of three columns with generated data containing stock purchase transactions: Name (string), Quantity (int) and Price (float)

1. Cruz, 700, 22.00
2. Rina, 77, 174.04
3. Caryn, 62, 667.50
4. Hop, 971, 342.37
5. Donovan, 406, 684.31
6. Caleb, 602, 310.12
7. Andrew, 789, 417.30
8. Blossom, 605, 992.47
9. Mitchel, 506, 31.03
10. Sharon, 647, 58.99

Figure 5: CSV file dataset snippet
Source: own representation
Implementation
Initial Effort

Three classes:

1. **RowStore:**
   - Row: `struct Row [Name, Quantity, Price]` object
   - Table: `vector<Row>` object containing such rows

2. **ColumnStore:**
   - Column: `vector<>` object depending on the column type
   - Table: `struct Table [vector(Name), vector(Quantity), vector(Price)]`

3. **HybridStore:**
   - Partial row: `struct MiniRow [Quantity, Price]` object
   - Table: `vector(Name)` object for the first column, and a `vector<MiniRow>` for the second “column” containing (Quantity, Price) per entry
Implementation
Initial Effort

Functions for each class:
• Insertion and selection
• $\text{SUM}(\text{Quantity})$
• $\text{AVG}(\text{Price})$
• $\text{SUM}(\text{Quantity} \times \text{Price})$
Table of contents

1. Introduction
2. Row Store
3. Column Store
4. Hybrid Store
5. **Implementation**
   1. Initial Effort
   2. **Evaluation**
   3. Approach
   4. Results
6. Questions
Implementation Evaluation

Theoretically accessing the data in three different ways

Once CSV file loaded, objects are stored in-memory to be used during runtime

Longer time for initial loading, but much shorter time to execute the aggregations

Testing with a dataset of 1000 rows almost unmeasurable

<table>
<thead>
<tr>
<th>Store Type</th>
<th># Entries</th>
<th>1 Million</th>
<th>10 Million</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Row Store</td>
<td>Column Store</td>
</tr>
<tr>
<td>INSERT</td>
<td></td>
<td>91929</td>
<td>101312</td>
</tr>
<tr>
<td>SELECT</td>
<td></td>
<td>3117</td>
<td>3444</td>
</tr>
<tr>
<td>SUM(Quantity)</td>
<td></td>
<td>212</td>
<td>718</td>
</tr>
<tr>
<td>AVG(Price)</td>
<td></td>
<td>204</td>
<td>215</td>
</tr>
<tr>
<td>SUM(Quantity*Price)</td>
<td></td>
<td>223</td>
<td>226</td>
</tr>
</tbody>
</table>

Figure 6: Execution times for 1 million and 10 million entries (in milliseconds)
Source: own representation
Table of contents

1. Introduction
2. Row Store
3. Column Store
4. Hybrid Store
5. Implementation
   1. Initial Effort
   2. Evaluation
   3. Approach
   4. Results
6. Questions
Implementation Approach

New approach by using already stored data
One CSV file for each store mode: row, column and hybrid
Data already prepared as would be expected for each store

Figure 7: Input CSV file structure for the second approach
Source: own representation
Implementation Approach

Two advantages using these CSV files as “storage”:
1. Slowed down access when performing aggregations because the content is not loaded in-memory but continuously from a slower storage
2. Minimized influence of the environment factors and further internal optimization when dealing with known objects

Operation:
No INSERT operation in this approach
SELECT(*) (with a filter on Name)
COUNT(*)
SUM(Quantity)
AVG(Price)
SUM(Quantity*Price)
Table of contents

1. Introduction
2. Row Store
3. Column Store
4. Hybrid Store
5. Implementation
   1. Initial Effort
   2. Evaluation
   3. Approach
4. Results
6. Questions
Implementation

Results

Longer time for aggregation operations

<table>
<thead>
<tr>
<th>Store Type</th>
<th>1 Million</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Row Store</td>
</tr>
<tr>
<td>SELECT * [Name='Zoe']</td>
<td>922</td>
</tr>
<tr>
<td>COUNT(*)</td>
<td>826</td>
</tr>
<tr>
<td>SUM(Quantity)</td>
<td>1482</td>
</tr>
<tr>
<td>AVG(Price)</td>
<td>4673</td>
</tr>
<tr>
<td>SUM(Quantity*Price)</td>
<td>8697</td>
</tr>
</tbody>
</table>

Figure 8: Second approach execution times for 1 million entries (in milliseconds)
Source: own representation

COUNT(*) – faster in column and hybrid store
SUM(Quantity) and AVG(Price) – faster in column store
SUM(Quantity*Price) – slightly faster in column store
SELECT – faster in column and hybrid store
# Table of contents

1. Introduction
2. Row Store
3. Column Store
4. Hybrid Store
5. Implementation
6. Questions
Questions?
References
