Main-Memory DBMSs for OLAP
Online Analytical Processing (OLAP)
Online Analytical Processing (OLAP)

read-mostly
Online Analytical Processing (OLAP)

analysis mostly on **few columns** and over **all rows**
Online Analytical Processing (OLAP)

query response time is important
Main-Memory DBMSs for OLAP

• read-mostly
  → Data is typically not modified or updated instead new data is just inserted
  → Append new data via bulk loading
Main-Memory DBMSs for OLAP

- **read-mostly**
  - Data is typically not modified or updated instead new data is just inserted
  - Append new data via bulk loading

- **analysis mostly on few columns and over all rows**
  - Use column-oriented data layout
  - Yields better data cache utilization
Main-Memory DBMSs for OLAP

• read-mostly
  → Data is typically not modified or updated instead new data is just inserted
  → Append new data via bulk loading

• analysis mostly on **few columns** and over **all rows**
  → Use column-oriented data layout
  → Yields better data cache utilization

• query **response time** is important
  → Bulk processing to improve instruction cache effectiveness
  → Reduce amount of data to process by
    • lightweight compression techniques
    • optimizing query processing
Main-Memory DBMSs for OLAP

- Main memory as primary storage
- Column-oriented data layout
- Lightweight compression
- Optimized query processing

→ Main-Memory DBMS for efficient OLAP
Main-Memory DBMSs for OLAP

- Main memory as primary storage ✓
- Column-oriented data layout ✓
- Lightweight compression ○
- Optimized query processing ○

~ Main-Memory DBMS for efficient OLAP
Data Compression - Motivation

Reduce size of data
Data Compression - Motivation

Reduce size of data

→ **Reduced costs** for storage as we need less storage space to store the same amount of data

→ **More data** can be stored using the same amount of storage space

→ Better utilization of **memory bandwidth**
Data Compression - Requirements

• Lossless compression → otherwise we generate data errors
• Lightweight (de-)compression → otherwise (de-)compression overhead would outweigh our possible performance potentials
• Enable processing of compressed values → no additional overhead for decompression
Data Compression - Requirements

• Lossless compression
  → otherwise we generate data errors
Data Compression - Requirements

- Lossless compression
  → otherwise we generate data errors

- Lightweight (de-)compression
  → otherwise (de-)compression overhead would outweigh our possible performance potentials
Data Compression - Requirements

- Lossless compression
  → otherwise we generate data errors

- Lightweight (de-)compression
  → otherwise (de-)compression overhead would outweigh our possible performance potentials

- Enable processing of compressed values
  → no additional overhead for decompression
Classification of compression techniques

• General idea: Replace data by representation that needs less bits than original data
Classification of compression techniques

• General idea: Replace data by representation that needs less bits than original data
• Granularity:
  • Attribute values, tuples, tables, pages
  • Index structures
Classification of compression techniques

- General idea: Replace data by representation that needs less bits than original data
- Granularity:
  - Attribute values, tuples, tables, pages
  - Index structures
- Code length:
  - **Fixed code length**: All values are encoded with same number of bits
  - **Variable code length**: Number of bits differs (e.g., correlate number of used bits with value frequency; Huffman Encoding)
Dictionary Encoding

- Use a dictionary that contains data values and their surrogates
- Surrogate can be derived from values’ dictionary position
- Applicable to row- and column-oriented data layouts

<table>
<thead>
<tr>
<th>...</th>
<th>State</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Thuringia</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thuringia</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Saxony</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sx. Anhalt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hesse</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bavaria</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hesse</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sx. Anhalt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thuringia</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>...</th>
<th>State</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td></td>
</tr>
</tbody>
</table>

Dictionary

<table>
<thead>
<tr>
<th>State</th>
<th>Surrogate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bavaria</td>
<td>0000</td>
</tr>
<tr>
<td>Hesse</td>
<td>0001</td>
</tr>
<tr>
<td>Saxony</td>
<td>0010</td>
</tr>
<tr>
<td>Sx. Anhalt</td>
<td>0011</td>
</tr>
<tr>
<td>Thuringia</td>
<td>0100</td>
</tr>
</tbody>
</table>
Bit Packing

• Surrogate values do not have to be multiples of one byte
• Example: 16 distinct values can be effectively stored using 4 bit per surrogate → 2 values per byte

→ Processing of compressed values is not straightforward
Run Length Encoding

- Reduce size of sequences of same value
- Store the value and an indicator about the sequence length
- Applicable to column-oriented data layouts
- Sorting can further improve compression effectiveness
Common Value Suppression

- A common value is scattered across a column (e.g., null)
- Use a data structure that indicates whether a common value is stored at a given row index or not
  - Yes: Common value is stored here
  - No: Lookup value in the dictionary (using prefix sum)
- Applicable to column-oriented data layouts

<table>
<thead>
<tr>
<th>...</th>
<th>State</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>Thuringia</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>...</td>
</tr>
<tr>
<td>Saxony</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>...</td>
</tr>
<tr>
<td>Hesse</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Bavaria</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>...</td>
</tr>
<tr>
<td>Sx. Anhalt</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Common Value

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Dictionary

<table>
<thead>
<tr>
<th>State</th>
<th>Surrogate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bavaria</td>
<td>0000</td>
</tr>
<tr>
<td>Hesse</td>
<td>0001</td>
</tr>
<tr>
<td>Saxony</td>
<td>0010</td>
</tr>
<tr>
<td>Sx. Anhalt</td>
<td>0011</td>
</tr>
<tr>
<td>Thuringia</td>
<td>0100</td>
</tr>
</tbody>
</table>

Exception Values
Bit-Vector Encoding

- Suitable for columns that have low number of distinct values
- Use bit string for every column value that indicates whether the value is present at a given row index or not
- Length of bit string equals number of tuples
- Used in Bitmap-Indexes

<table>
<thead>
<tr>
<th>Customer Status</th>
<th>Platinum</th>
<th>Gold</th>
<th>Silver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platinum</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Silver</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Platinum</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Gold</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Gold</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Silver</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Platinum</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Gold</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Silver</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Delta Coding

• Store difference to precedent value instead of the original value
• Applicable to column-oriented data layouts
• Sorting can further improve compression effectiveness

<table>
<thead>
<tr>
<th>...</th>
<th>Postal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>39104</td>
</tr>
<tr>
<td></td>
<td>39106</td>
</tr>
<tr>
<td></td>
<td>39108</td>
</tr>
<tr>
<td></td>
<td>39130</td>
</tr>
<tr>
<td></td>
<td>80336</td>
</tr>
<tr>
<td></td>
<td>80339</td>
</tr>
<tr>
<td></td>
<td>80807</td>
</tr>
<tr>
<td></td>
<td>80809</td>
</tr>
<tr>
<td></td>
<td>80933</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>...</th>
<th>Postal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>39104</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>41206</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>468</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Frequency Compression

- Idea: Exploit data skew
- Principle:
  - More frequent values are encoded using fewer bits
  - Less frequent values are encoded using more bits
- Use prefix codes (e.g., Huffman Encoding)
Frequency Compression - Column Store

Keep track where encoded values start and end
→ Use fixed code length for a partition rather than unique code for every value (i.e., Huffman Encoding)

Region 1
Bank 1: 3 bit tuplets in 128-bit words, with 2 bits of padding

Region 2
Bank 1: 8 bit tuplets in 64-bit words (no padding)

Tuple Map
101001000... (1st, 3rd, 6th entries are in region 1)

Picture taken from [Raman et al., 2013]
Frequency Compression - Row Store

Apply frequency compression on each column and perform additional delta coding

→ Introduces overhead for reading $n$th column value (2 ns per column value)

Picture taken from [Raman and Swart, 2006]
Frequency Partitioning

- Developed for IBM’s BLINK project [Raman et al., 2008]
- Similar to frequency compression of tuples in a row-oriented data layout
- But, partitioning tuples regarding column values
  → Overhead is reduced as within one partition code length is fixed
Frequency Partitioning: Principle

Picture taken from [Raman et al., 2008]
Data Compression - Summary

• General idea: Replace data by representation that needs less bits than original data

• Discussed approaches:
  • **Fixed code length**: Dictionary Encoding, RLE, Common Value Suppression
  • **Variable code length**: Delta Coding, Frequency Compression, Frequency Partitioning

• Improvements: bit packing, partitioning

• Benefits for main-memory DBMSs:
  • Reduced storage requirements
  • Better memory bandwidth utilization
## Compression in Action

<table>
<thead>
<tr>
<th>#</th>
<th>Table</th>
<th>Column</th>
<th>In-memory size in MiB</th>
<th>Compression Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>MonetDB</td>
<td>SAP HANA</td>
</tr>
<tr>
<td>1</td>
<td>HG00096</td>
<td>readid</td>
<td>4,122.000</td>
<td>2,662.777</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>refid</td>
<td>4,122.000</td>
<td>4,495.983</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>insert_offset</td>
<td>4,122.000</td>
<td>0.076</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>base</td>
<td>1,030.635</td>
<td>0.771</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>base_call_quality</td>
<td>4,122.000</td>
<td>1.758</td>
</tr>
<tr>
<td>6</td>
<td>grch37</td>
<td>id</td>
<td>950.875</td>
<td>2,614.746</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>position</td>
<td>950.875</td>
<td>1,782.782</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>base</td>
<td>237.875</td>
<td>0.003</td>
</tr>
<tr>
<td>9</td>
<td>reads</td>
<td>id</td>
<td>43.125</td>
<td>107.552</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>mapping_quality</td>
<td>43.125</td>
<td>0.010</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>overall</td>
<td>19,744.510</td>
<td>11,666.458</td>
</tr>
</tbody>
</table>
## Compression in Action

<table>
<thead>
<tr>
<th>#</th>
<th>Table</th>
<th>Column</th>
<th>In-memory size in MiB</th>
<th>Compression Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HG00096</td>
<td>readid</td>
<td>4,122.000</td>
<td>64.599</td>
</tr>
<tr>
<td>2</td>
<td>HG00096</td>
<td>refid</td>
<td>4,122.000</td>
<td>109.073</td>
</tr>
<tr>
<td>3</td>
<td>HG00096</td>
<td>insert_offset</td>
<td>4,122.000</td>
<td>0.002</td>
</tr>
<tr>
<td>4</td>
<td>HG00096</td>
<td>base</td>
<td>1,030.635</td>
<td>0.075</td>
</tr>
<tr>
<td>5</td>
<td>HG00096</td>
<td>base_call_quality</td>
<td>4,122.000</td>
<td>0.043</td>
</tr>
<tr>
<td>6</td>
<td>grch37</td>
<td>id</td>
<td>950.875</td>
<td>274.983</td>
</tr>
<tr>
<td>7</td>
<td>grch37</td>
<td>position</td>
<td>950.875</td>
<td>187.489</td>
</tr>
<tr>
<td>8</td>
<td>grch37</td>
<td>base</td>
<td>237.875</td>
<td>0.001</td>
</tr>
<tr>
<td>9</td>
<td>reads</td>
<td>id</td>
<td>43.125</td>
<td>249.396</td>
</tr>
<tr>
<td>10</td>
<td>reads</td>
<td>mapping_quality</td>
<td>43.125</td>
<td>0.024</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>overall</td>
<td>19,744.510</td>
<td>59.087</td>
</tr>
</tbody>
</table>
## Compression in Action

<table>
<thead>
<tr>
<th>#</th>
<th>Table</th>
<th>Column</th>
<th>In-memory size in MiB</th>
<th>Compression Ratio in %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>MonetDB</td>
<td>SAP HANA</td>
</tr>
<tr>
<td>1</td>
<td>HG00096</td>
<td>readid</td>
<td>4,122.000</td>
<td>2,662.777</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>refid</td>
<td>4,122.000</td>
<td>4,495.983</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>insert_offset</td>
<td>4,122.000</td>
<td>0.076</td>
</tr>
<tr>
<td>4</td>
<td>HG00096</td>
<td>base</td>
<td>1,030.635</td>
<td>0.771</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>base_call_quality</td>
<td>4,122.000</td>
<td>1.758</td>
</tr>
<tr>
<td>6</td>
<td>grch37</td>
<td>id</td>
<td>950.875</td>
<td>2,614.746</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>position</td>
<td>950.875</td>
<td>1,782.782</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>base</td>
<td>237.875</td>
<td>0.003</td>
</tr>
<tr>
<td>9</td>
<td>reads</td>
<td>id</td>
<td>43.125</td>
<td>107.552</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>mapping_quality</td>
<td>43.125</td>
<td>0.010</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>overall</td>
<td>19,744.510</td>
<td>11,666.458</td>
</tr>
</tbody>
</table>
Main-Memory DBMSs for OLAP

Main memory as primary storage  ✓
+ Column-oriented data layout  ✓
+ Lightweight compression  ✓
+ Optimized query processing  ❌

Main-Memory DBMS for efficient OLAP
Query Processing - Motivation

- For analytical applications (OLAP), most of the time a small set of columns is needed to answer a query
  → Read only columns that are needed to answer a query
  → Column-oriented data layout is highly suitable
- Beware for queries accessing all columns of a table
  → high materialization costs
Materialization Strategies

• Recall: In a column-oriented data layout each column is stored separately

• Consider, e.g., a selection query:

```sql
SELECT l_shipdate, l_linenumber
FROM lineitem
WHERE l_shipdate < "2009-09-26"
     AND l_linenumber < "10000"
```

→ Query accesses and returns values of two columns
→ Materialization (tuple reconstruction) during query processing necessary
Early Materialization

Reconstruct tuples as soon as possible

Picture taken from [Abadi et al., 2007]

SPC ... Scan, Predicate, Construct
Late Materialization

Postpone tuple reconstruction to the latest possible time

![Diagram of late materialization](image)

Picture taken from [Abadi et al., 2007]

DS ... Data Sources
Advantages of Early and Late Materialization

Early Materialization (EM):

- Reduces access cost if one column has to be accessed multiple times during query processing
- ↑ [Abadi et al., 2007]

Late Materialization (LM):

- Reduces amount of tuples to reconstruct
- LM allows processing of columns as long as possible
  → Processing of compressed data
  → LM improves cache effectiveness
- ↑ [Abadi et al., 2008]
Example: Invisible Join  [Abadi et al., 2008]

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;
Example: Invisible Join  

[Abadi et al., 2008]

```
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;
```
Example: Invisible Join - Hash [Abadi et al., 2008]

### Apply region = 'Asia' on Customer table

| custkey | region | nation | ...
|---------|--------|--------|-----
| 1       | Asia   | China  |     |
| 2       | Europe | France |     |
| 3       | Asia   | India  |     |

### Apply region = 'Asia' on Supplier table

| suppkey | region | nation | ...
|---------|--------|--------|-----
| 1       | Asia   | Russia |     |
| 2       | Europe | Spain  |     |

### Apply year in [1992, 1997] on Date table

| dateid     | year | ...
|------------|------|-----
| 01011997   | 1997 |     |
| 01021997   | 1997 |     |
| 01031997   | 1997 |     |

Hash table with keys 1 and 3

Hash table with key 1

Hash table with keys 01011997, 01021997, and 01031997
Example: Invisible Join  [Abadi et al., 2008]

```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;
```
Example: Invisible Join - Probe  [Abadi et al., 2008]

Fact Table

<table>
<thead>
<tr>
<th>orderkey</th>
<th>custkey</th>
<th>suppkey</th>
<th>orderdate</th>
<th>revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>01011997</td>
<td>43256</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>01011997</td>
<td>33333</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>01021997</td>
<td>12121</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>01021997</td>
<td>23233</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>01021997</td>
<td>45456</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>01031997</td>
<td>43251</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>2</td>
<td>01031997</td>
<td>34235</td>
</tr>
</tbody>
</table>

probe

Hash table with keys 1 and 3

1
1
0
1
1
0
1
1

matching fact table bitmap for cust. dim. join

Hash table with key 1

1
0
1
1
1
0
0
0

Bitwise And

1
0
1
0
0
0
0
0

Hash table with keys 01011997, 01021997, and 01031997

1
1
1
1
1
1
1
1

fact table tuples that satisfy all join predicates
Example: Invisible Join  [Abadi et al., 2008]

```
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;
```
Example: Invisible Join - Materialize [Abadi et al., 2008]
Star-Schema-Benchmark Performance

- T = tuple-at-a-time processing, t = block processing
- I = invisible join enabled, i = disabled
- C = compression enabled, c = disabled
- L = late materialization enabled, l = disabled

Figure taken from [Abadi et al., 2008]
Impact of column-store optimizations (ranked according to previous experiment)

1. Late materialization
   \[\rightarrow\] process only needed data

2. Compression
   \[\rightarrow\] better memory bandwidth utilization

3. Bulk processing
   \[\rightarrow\] improves instruction-cache effectiveness

4. Invisible Join
   \[\rightarrow\] optimized join processing that makes use of late materialization
Main-Memory DBMSs for OLAP

- Main memory as primary storage ✓
- Column-oriented data layout ✓
- Lightweight compression ✓
- Optimized query processing ✓

→ Main-Memory DBMS for efficient OLAP
Main-Memory DBMSs for OLTP

Partially based on Transaction Management in Main-Memory DBMSs by Sebastian Breß from TU Dortmund.
Online Transaction Processing (OLTP)
Online Transaction Processing (OLTP)

mixed workloads, read and write
Online Transaction Processing (OLTP)

query mostly over all columns but only a few rows
Online Transaction Processing (OLTP)

query throughput is important
Main-Memory DBMSs for OLTP

- mixed workloads, read and write
  → Data layout needed that allows for fast updates and inserts
Main-Memory DBMSs for OLTP

- mixed workloads, **read and write**
  → Data layout needed that allows for fast updates and inserts
- query mostly over **all columns** but only a **few rows**
  → Use row-oriented data layout
  → Yields better data cache utilization
Main-Memory DBMSs for OLTP

- Mixed workloads, read and write
  → Data layout needed that allows for fast updates and inserts
- Query mostly over all columns but only a few rows
  → Use row-oriented data layout
  → Yields better data cache utilization
- Query throughput is important
  → Fully exploit modern hardware by minimizing functional overhead introduced by traditional, disk-based DBMSs
  - Single-threaded vs. multi-threaded execution
  - Logging vs. high availability
Single-Threaded vs. Multi-Threaded Execution

- System R like systems use multi-threading to hide disk latencies and to increase throughput

- Required functionality
  1. Ensure serializability of transactions
     → Use locking
  2. Protect data structures for concurrent access
     → Use latching

- Additional mechanisms introduce significant overhead
Single-Threaded vs. Multi-Threaded Execution (2)

- A single transaction’s execution time is roughly in the order of hundreds of microseconds

- If TXN are executed serially, we:
  1. Can remove locking, because serializability is guaranteed
  2. Can remove latching, because data structures are not accessed concurrently

→ Execute one transaction at a time is often faster!
Logging vs. High Availability

- Writing a log file introduces significant overhead

- Failover/rebuild is as efficient as using a redo log for recovery

  [Lau and Madden, 2006]

  1. Undo only required for running TXNs, undo log can be held in main-memory structure

  2. No need for redo, because of network recovery from remote site

    → If node restarts after system failure, the database is updated with data from the operational node

    → No persistent log file needed!
Overhead Breakdown of RDBMS Shore

Picture taken from [Harizopoulos et al., 2008]
Overhead Breakdown of RDBMS Shore: Payment TXN of TPC-C Benchmark

Picture taken from [Harizopoulos et al., 2008]
Alternative Design Considerations – Summary

• System R like systems introduce a lot of overhead if data can be kept in memory [Harizopoulos et al., 2008]:
  1. Logging
  2. Locking
  3. Latching
  4. Buffer management

• A redesigned engine such as H-Store can outperform traditional, system R like RDBMSs
  → Stonebraker and others observed a performance improvement by a factor of 82 [Stonebraker et al., 2007]
H-Store

• Next-generation OLTP system

• Operates on a distributed cluster of shared-nothing machines

• Data resides entirely in main memory

• Row-oriented storage
H-Store

• H-Store site: basic operational entity, single-threaded execution of TXNs, assigned to one core
  → Sites independent from each other
  → No sharing of data structures or memory with co-located sites running on the same machine

• H-Store node: physical machine, hosts multiple sites
  (≈#CPU cores)

• H-Store instance: cluster of two or more nodes
H-Store: Architecture

Deployment Framework
- Database Schema
- Cluster Information
- Stored Procedures
- Sample Workload
- Database Designer
- Query Planner/Optimizer
- Compiled Stored Procedures
- Query Plans
- Physical Layout

Runtime Time
- OLTP Application
  - H-Store API
  - Transaction Initiator
  - Messaging Fabric
  - Other Cluster Execution Nodes
  - Main Memory Storage Manager
  - Stored Procedure Executor
  - Query Execution Engine
  - System Catalogs

Picture taken from [Kallman et al., 2008]
Main-Memory DBMSs for OLTP and OLAP
Main-Memory DBMSs for OLTP and OLAP

Combining OLTP and OLAP in one system leads to conflicting requirements!
Main-Memory DBMSs for OLTP and OLAP

read-mostly ≠ read AND write
Main-Memory DBMSs for OLTP and OLAP

query over
FEW columns but ALL rows ∨ ALL columns but FEW rows
Main-Memory DBMSs for OLTP and OLAP

query over

query response time $\lesssim$ query throughput
Design Considerations

• Data layout:
  → Which data layout to use for storing data?

• Data processing:
  → How to process analytical and transactional workloads simultaneously?
Data Layout

Different data stores are advantageous for different database application

- Column stores are advantageous for OLAP workloads
  → Aggregation over complete columns, just read needed data
- Row stores are advantageous for OLTP workloads
  → Update, write, record access
Data Layout

Different data stores are advantageous for different database application

- Column stores are advantageous for OLAP workloads
  → Aggregation over complete columns, just read needed data
- Row stores are advantageous for OLTP workloads
  → Update, write, record access

→ Single data layouts seem not capable for OLTP and OLAP processing at the same time
→ Use a hybrid data layout/storage engine:

1. Storage Advisor [Rösch et al., 2012]
2. Delta Store [Sikka et al., 2012]
Storage Advisor for Hybrid-Store Databases [Rösch et al., 2012]

- Idea: Cost-based recommendation of data layout based on
  - Workload (online and offline)
  - Data schema
  - Data characteristics
- Goal: Minimize runtime of given (offline) workload
Storage Advisor for Hybrid-Store Databases [Rösch et al., 2012]

- Idea: Cost-based recommendation of data layout based on:
  - Workload (online and offline)
  - Data schema
  - Data characteristics
- Goal: Minimize runtime of given (offline) workload
Storage Advisor for Hybrid-Store Databases [Rösch et al., 2012]

Recommendations are made for complete tables
Recommendations are made for complete tables but also for rows and columns.
Delta Store  [Sikka et al., 2012]

- Idea: Store records according to their life cycle stage:
  - New records → row store
  - Old records → column store
- Concrete storage layout is transparent to database operations
Delta Store [Sikka et al., 2012]

- Idea: Store records according to their life cycle stage:
  - New records → row store
  - Old records → column store
- Concrete storage layout is transparent to database operations
Data Processing

- OLAP and OLTP applications have different optimization goals → Response time vs. throughput
- Problems when processing both workloads at the same time:
  - OLAP queries are postponed until OLTP queries
    → ⫷ Response time
  - OLTP queries have to wait for long lasting OLAP queries
    → ⫷ query throughput

→ Decouple workloads in one DBMS!
HyPer [Kemper and Neumann, 2011]

- Supports efficient hybrid OLTP/OLAP processing
- Keeps all data in main memory
- OLTP queries run in one process, OLAP queries run on a virtual memory snapshot from the OLTP process
- Create snapshot using the fork system call of the Linux kernel
- Assumption: We do not need the latest consistent state for OLAP, because analysis result is hardly influenced by a couple of updates
HyPer: Copy on Write Mechanism

When a OLTP query attempts to modify a page:

- The linux kernel creates a new page containing the old data for the OLAP process → transparently to the DBMS!
- Then, the OLTP process can modify the page → OLAP process always has a transaction consistent snapshot
HyPer: Database Architecture

As efficient as dedicated OLTP main memory DBMS (e.g., VoltDB, TimesTen)

As fast as dedicated OLAP main memory DBMS (e.g., MonetDB, TRELX)

Picture taken from [Kemper and Neumann, 2011]
HyPer: Concept for Hybrid OLTP/OLAP

Picture taken from [Kemper and Neumann, 2011]
HyPer: Concept for Hybrid OLTP/OLAP

Picture taken from [Kemper and Neumann, 2011]
Summary

• Rethinking the architecture of DBMSs to adapt them on changes in hardware pays off [Harizopoulos et al., 2008]

• Modern transactional systems keep their operational data in main memory
  → We have to optimize for different things than 30 years ago

→ New architectures mean that we have to solve old problems again, but for the current hardware landscape and the applications of today and tomorrow!
References I

Column-stores vs. row-stores: how different are they really?
In *SIGMOD*, pages 967–980.

Materialization Strategies in a Column-Oriented DBMS.
In *ICDE*, pages 466–475.

Monetdb/x100: Hyper-pipelining query execution.
In *CIDR*, pages 225–237.

A decomposition storage model.
In *SIGMOD*, pages 268–279.

What Every Programmer Should Know About Memory.

Encapsulation of Parallelism in the Volcano Query Processing System.
In *SIGMOD*, pages 102–111.
References II

OLTP through the looking glass, and what we found there.
In *SIGMOD*, pages 981–992.

*Computer Architecture: A Quantitative Approach.*
Morgan Kaufmann, 2 edition.

*Computer Architecture: A Quantitative Approach.*
Morgan Kaufmann, 4 edition.

*Computer Architecture - A Quantitative Approach.*
Morgan Kaufmann, 5 edition.

Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S.,

HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots.
References III

An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse.
In VLDB, pages 703–714.


Database Architecture Evolution: Mammals flourished long before Dinosaurs became extinct.

DB2 with BLU Acceleration: So Much More Than Just a Column Store.
PVLDB, 6(11):1080–1091.

How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations.
In VLDB, pages 858–869.

Constant-Time Query Processing.
In ICDE, pages 60–69.
References IV


