Main-Memory
Database Management Systems

Andreas Meister
Otto-von-Guericke University Magdeburg
Summer Term 2017
We will talk about

Computer and Database Systems Architecture ✓

Cache Awareness ✓

Processing Models

Storage Models
Processing Models

Or how to improve the instruction cache effectiveness?
Processing Models

There are basically two alternative processing models that are used in modern DBMSs:

- **Tuple-at-a-time volcano model** [Graefe, 1990]
  - Operator requests next tuple, processes it, and passes it to the next operator

- **Operator-at-a-time bulk processing** [Manegold et al., 2009]
  - Operator consumes its input and materializes its output
Tuple-At-A-Time Processing

Most systems implement the **Volcano iterator model**:

- Operators request tuples from their input using \texttt{next}().
- Data is processed \textbf{tuple at a time}.
- Each operator keeps its own state.

```
Operator 1
next ()  tuple
Operator 2
next ()  tuple
Operator 3
next ()  tuple
...  ...
```

Database
Tuple-At-A-Time Processing - Consequences

• Pipeline-parallelism
  → Data processing can start although data does not fully reside in main memory
  → Small intermediate results
Tuple-At-A-Time Processing - Consequences

- **Pipeline-parallelism**
  - Data processing can start although data does not fully reside in main memory
  - Small intermediate results

- All operators in a plan run **tightly interleaved**.
  - Their combined instruction footprint may be large.
  - Instruction cache misses.
Tuple-At-A-Time Processing - Consequences

- **Pipeline-parallelism**
  - Data processing can start although data does not fully reside in main memory
  - Small intermediate results

- All operators in a plan run **tightly interleaved**.
  - Their combined instruction footprint may be large.
  - Instruction cache misses.

- Operators constantly call each other’s functionality.
  - Large function call overhead.
Tuple-At-A-Time Processing - Consequences

- Pipeline-parallelism
  - Data processing can start although data does not fully reside in main memory
  - Small intermediate results

- All operators in a plan run tightly interleaved.
  - Their combined instruction footprint may be large.
  - Instruction cache misses.

- Operators constantly call each other’s functionality.
  - Large function call overhead.

- The combined state may be too large to fit into caches.
  - E.g., hash tables, cursors, partial aggregates.
  - Data cache misses.
Example: TPC-H Query Q1 on MySQL

```
SELECT l_returnflag, l_linestatus, SUM(l_quantity) AS sum_qty,
     SUM(l_extendedprice) AS sum_base_price,
     SUM(l_extendedprice*(1-l_discount)) AS sum_disc_price,
     SUM(l_extendedprice*(1-l_discount)*(1+l_tax)) AS sum_charge,
     AVG(l_quantity) AS avg_qty, AVG(l_extendedprice) AS avg_price,
     AVG(l_discount) AS avg_disc, COUNT(*) AS count_order
FROM lineitem
WHERE l_shipdate <= DATE '1998-09-02'
GROUP BY l_returnflag, l_linestatus
```

- **Scan query** with **arithmetics** and a bit of aggregation.

Source: MonetDB/X100: Hyper-Pipelining Query Execution.

[Boncz et al., 2005]
<table>
<thead>
<tr>
<th>time [sec]</th>
<th>calls</th>
<th>instr./call</th>
<th>IPC</th>
<th>function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.9</td>
<td>846M</td>
<td>6</td>
<td>0.64</td>
<td>ut_fold_uint_pair</td>
</tr>
<tr>
<td>8.5</td>
<td>0.15M</td>
<td>27K</td>
<td>0.71</td>
<td>ut_fold_binary</td>
</tr>
<tr>
<td>5.8</td>
<td>77M</td>
<td>37</td>
<td>0.85</td>
<td>memcpy</td>
</tr>
<tr>
<td>3.1</td>
<td>23M</td>
<td>64</td>
<td>0.88</td>
<td>Item_sum_sum::update_field</td>
</tr>
<tr>
<td>3.0</td>
<td>6M</td>
<td>247</td>
<td>0.83</td>
<td>row_search_for_mysql</td>
</tr>
<tr>
<td>2.9</td>
<td>17M</td>
<td>79</td>
<td>0.70</td>
<td>Item_sum_avg::update_field</td>
</tr>
<tr>
<td>2.6</td>
<td>108M</td>
<td>11</td>
<td>0.60</td>
<td>rec_get_bit_field_1</td>
</tr>
<tr>
<td>2.5</td>
<td>6M</td>
<td>213</td>
<td>0.61</td>
<td>row_sel_store_mysql_rec</td>
</tr>
<tr>
<td>2.4</td>
<td>48M</td>
<td>25</td>
<td>0.52</td>
<td>rec_get_nth_field</td>
</tr>
<tr>
<td>2.4</td>
<td>60</td>
<td>19M</td>
<td>0.69</td>
<td>ha_print_info</td>
</tr>
<tr>
<td>2.4</td>
<td>5.9M</td>
<td>195</td>
<td>1.08</td>
<td>end_update</td>
</tr>
<tr>
<td>2.1</td>
<td>11M</td>
<td>89</td>
<td>0.98</td>
<td>field_conv</td>
</tr>
<tr>
<td>2.0</td>
<td>5.9M</td>
<td>16</td>
<td>0.77</td>
<td>Field_float::val_real</td>
</tr>
<tr>
<td>1.8</td>
<td>5.9M</td>
<td>14</td>
<td>1.07</td>
<td>Item_field::val</td>
</tr>
<tr>
<td>1.5</td>
<td>42M</td>
<td>17</td>
<td>0.51</td>
<td>row_sel_field_store_in_mysql</td>
</tr>
<tr>
<td>1.4</td>
<td>36M</td>
<td>18</td>
<td>0.76</td>
<td>buf_frame_align</td>
</tr>
<tr>
<td>1.3</td>
<td>17M</td>
<td>38</td>
<td>0.80</td>
<td>Item_func_mul::val</td>
</tr>
<tr>
<td>1.4</td>
<td>25M</td>
<td>25</td>
<td>0.62</td>
<td>pthread_mutex_unlock</td>
</tr>
<tr>
<td>1.2</td>
<td>206M</td>
<td>2</td>
<td>0.75</td>
<td>hash_get_nth_cell</td>
</tr>
<tr>
<td>1.2</td>
<td>25M</td>
<td>21</td>
<td>0.65</td>
<td>mutex_test_and_set</td>
</tr>
<tr>
<td>1.0</td>
<td>102M</td>
<td>4</td>
<td>0.62</td>
<td>rec_get_1byte_offs_flag</td>
</tr>
<tr>
<td>1.0</td>
<td>53M</td>
<td>9</td>
<td>0.58</td>
<td>rec_1_get_field_start_offs</td>
</tr>
<tr>
<td>0.9</td>
<td>42M</td>
<td>11</td>
<td>0.65</td>
<td>rec_get_nth_fieldExtern_bit</td>
</tr>
<tr>
<td>1.0</td>
<td>11M</td>
<td>38</td>
<td>0.80</td>
<td>Item_func_minus::val</td>
</tr>
<tr>
<td>0.5</td>
<td>5.9M</td>
<td>38</td>
<td>0.80</td>
<td>Item_func_plus::val</td>
</tr>
</tbody>
</table>
Observations

• Only **single tuple** processed in each call; **millions of calls**.
• Only **10 % of the time** spent on actual query task.
• Low **instructions-per-cycle** (IPC) ratio.

\(^1\) Depends on underlying hardware
Observations

- Only **single tuple** processed in each call; **millions of calls**.
- Only **10 % of the time** spent on actual query task.
- Low **instructions-per-cycle** (IPC) ratio.

- Much time spent on **field access** (e.g., `rec_get_nth_field()`).
  - Polymorphic operators
- Single-tuple functions hard to optimize (by compiler).
  → Low instructions-per-cycle ratio.
  → Vector instructions (SIMD) hardly applicable.
- Function call overhead (e.g., `Item_func_plus::val()`).
  - \[
  \frac{38 \text{ instr.}}{0.8 \text{ instr./cycle}} = 48 \text{ cycles vs. } 3 \text{ instr. for load/add/store assembly}^1
  \]

^1 Depends on underlying hardware
Operator-At-A-Time Processing

- Operators consume and produce full tables.
- Each (sub-)result is **fully materialized** (in memory).
- **No** pipelining (rather a sequence of statements).
- Each operator runs exactly once.
Operator-At-A-Time Processing

Function call overhead is now replaced by extremely tight loops.

Example: batval_int_add(···)

```c
if (vv != int_nil) {
    for (; bp < bq; bp++, bnp++) {
        REGISTER int bv = *bp;
        if (bv != int_nil) {
            bv = (int) OP(bv,+,vv);
        }
        *bnp = bv;
    }
} else {
    ...
}
```
Operator-At-A-Time Consequences

- Parallelism: **Inter-operator** and **intra-operator**

  - Function call overhead is now replaced by extremely tight loops that conveniently fit into instruction caches, can be optimized effectively by modern compilers → loop unrolling → vectorization (use of SIMD instructions)
  - Function calls are now out of the critical code path.
  - No per-tuple field extraction or type resolution.
  - Operator specialization, e.g., for every possible type.
  - Implemented using macro expansion.
  - Possible due to column-based storage.
Operator-At-A-Time Consequences

- Parallelism: **Inter-operator** and **intra-operator**
- Function call overhead is now replaced by **extremely tight loops** that
  - conveniently **fit into instruction caches**,
  - can be **optimized** effectively by modern compilers
    - → loop unrolling
    - → vectorization (use of SIMD instructions)
  - can leverage modern CPU features (**hardware prefetching**).
Operator-At-A-Time Consequences

• Parallelism: **Inter-operator** and **intra-operator**
• Function call overhead is now replaced by extremely tight loops that
  • conveniently **fit into instruction caches**,  
  • can be **optimized** effectively by modern compilers
    → **loop unrolling**
    → **vectorization** (use of SIMD instructions)
  • can leverage modern CPU features (**hardware prefetching**).

• Function calls are now **out of the critical code path**.
Operator-At-A-Time Consequences

• Parallelism: **Inter-operator** and **intra-operator**

• Function call overhead is now replaced by **extremely tight loops** that
  
  • conveniently **fit into instruction caches**,  
  • can be **optimized** effectively by modern compilers  
    → **loop unrolling**  
    → **vectorization** (use of SIMD instructions)  
  • can leverage modern CPU features (**hardware prefetching**).

• Function calls are now **out of the critical code path**.

• **No** per-tuple field extraction or type resolution.
  
  • **Operator specialization**, *e.g.*, for every possible type.
  • Implemented using **macro expansion**.
  • Possible due to column-based storage.
<table>
<thead>
<tr>
<th>result size</th>
<th>time [ms]</th>
<th>bandwidth [MB/s]</th>
<th>MIL statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.9M</td>
<td>127</td>
<td>352</td>
<td>$s_0 := \text{select (l_shipdate, \ldots).mark ();}$</td>
</tr>
<tr>
<td>5.9M</td>
<td>134</td>
<td>505</td>
<td>$s_1 := \text{join (s_0, l_returnag);}$</td>
</tr>
<tr>
<td>5.9M</td>
<td>134</td>
<td>506</td>
<td>$s_2 := \text{join (s_0, l_linestatus);}$</td>
</tr>
<tr>
<td>5.9M</td>
<td>235</td>
<td>483</td>
<td>$s_3 := \text{join (s_0, l_extprice);}$</td>
</tr>
<tr>
<td>5.9M</td>
<td>233</td>
<td>488</td>
<td>$s_4 := \text{join (s_0, l_discount);}$</td>
</tr>
<tr>
<td>5.9M</td>
<td>232</td>
<td>489</td>
<td>$s_5 := \text{join (s_0, l_tax);}$</td>
</tr>
<tr>
<td>5.9M</td>
<td>134</td>
<td>507</td>
<td>$s_6 := \text{join (s_0, l_quantity);}$</td>
</tr>
<tr>
<td>5.9M</td>
<td>290</td>
<td>155</td>
<td>$s_7 := \text{group (s_1);}$</td>
</tr>
<tr>
<td>5.9M</td>
<td>329</td>
<td>136</td>
<td>$s_8 := \text{group (s_7, s_2);}$</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>$s_9 := \text{unique (s_8.mirror ());}$</td>
</tr>
<tr>
<td>5.9M</td>
<td>206</td>
<td>440</td>
<td>$r_0 := [+](1.0, s_5);$</td>
</tr>
<tr>
<td>5.9M</td>
<td>210</td>
<td>432</td>
<td>$r_1 := [-](1.0, s_4);$</td>
</tr>
<tr>
<td>5.9M</td>
<td>274</td>
<td>498</td>
<td>$r_2 := [*](s_3, r_1);$</td>
</tr>
<tr>
<td>5.9M</td>
<td>274</td>
<td>499</td>
<td>$r_3 := [*](s_12, r_0);$</td>
</tr>
<tr>
<td>4</td>
<td>165</td>
<td>271</td>
<td>$r_4 := {\text{sum}}(r_3, s_8, s_9);$</td>
</tr>
<tr>
<td>4</td>
<td>165</td>
<td>271</td>
<td>$r_5 := {\text{sum}}(r_2, s_8, s_9);$</td>
</tr>
<tr>
<td>4</td>
<td>163</td>
<td>275</td>
<td>$r_6 := {\text{sum}}(s_3, s_8, s_9);$</td>
</tr>
<tr>
<td>4</td>
<td>163</td>
<td>275</td>
<td>$r_7 := {\text{sum}}(s_4, s_8, s_9);$</td>
</tr>
<tr>
<td>4</td>
<td>144</td>
<td>151</td>
<td>$r_8 := {\text{sum}}(s_6, s_8, s_9);$</td>
</tr>
<tr>
<td>4</td>
<td>112</td>
<td>196</td>
<td>$r_9 := {\text{count}}(s_7, s_8, s_9);$</td>
</tr>
</tbody>
</table>

Source: MonetDB/X100: Hyper-Pipelining Query Execution.

[Boncz et al., 2005]
Tuple-At-A-Time vs. Operator-At-A-Time

The operator-at-a-time model is a two-edged sword:

- 😊 Cache-efficient with respect to code and operator state.
- 😊 Tight loops, optimizable code.
Tuple-At-A-Time vs. Operator-At-A-Time

The operator-at-a-time model is a two-edged sword:

- **😊** Cache-efficient with respect to code and operator state.
- **😊** Tight loops, optimizable code.

- ** вал** Data won’t fully fit into cache.
  - ➔ Repeated scans will fetch data from memory over and over.
  - ➔ Strategy falls apart when intermediate results no longer fit into main memory.
Tuple-At-A-Time vs. Operator-At-A-Time

The **operator-at-a-time model** is a two-edged sword:

- 🌟 Cache-efficient with respect to **code** and **operator state**.
- 🌟 Tight loops, optimizable code.

- 😞 **Data** won’t fully fit into cache.
  - → Repeated scans will fetch data **from memory** over and over.
  - → Strategy falls apart when intermediate results no longer fit into main memory.

Can we aim for the **middle ground** between the two extremes?

- tuple-at-a-time ← vectorized execution → operator-at-a-time
Vectorized Execution Model

Idea:

• Use Volcano-style iteration,

but:

• for each next() call return a large number of tuples
  → a so called “vector”
Vectorized Execution Model

Idea:

- Use Volcano-style iteration,

but:

- for each `next()` call return a large number of tuples
  → a so called “vector”

Choose vector size

- large enough to compensate for iteration overhead (function calls, instruction cache misses, ...), but
- small enough to not thrash data caches.
Vector Size ↔ Instruction Cache Effectiveness

- Vectorized execution quickly compensates for iteration overhead.
- 1000 tuples should conveniently fit into caches.

Source: [Zukowski, 2009]
## Comparison of Execution Models

<table>
<thead>
<tr>
<th>execution model</th>
<th>tuple</th>
<th>operator</th>
<th>vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>instr. cache utilization</td>
<td>poor</td>
<td>extremely good</td>
<td>very good</td>
</tr>
<tr>
<td>function calls</td>
<td>many</td>
<td>extremely few</td>
<td>very few</td>
</tr>
<tr>
<td>attribute access</td>
<td>complex</td>
<td>direct</td>
<td>direct</td>
</tr>
<tr>
<td>most time spent on</td>
<td>interpretation</td>
<td>processing</td>
<td>processing</td>
</tr>
<tr>
<td>CPU utilization</td>
<td>poor</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>compiler optimizations</td>
<td>limited</td>
<td>applicable</td>
<td>applicable</td>
</tr>
<tr>
<td>materialization overhead</td>
<td>very cheap</td>
<td>expensive</td>
<td>cheap</td>
</tr>
<tr>
<td>scalability</td>
<td>good</td>
<td>limited</td>
<td>good</td>
</tr>
</tbody>
</table>

Source [Zukowski, 2009]
Storage Models

Or how to improve the data cache effectiveness?
Data Storage Approaches

There are basically two alternative storage models that are used in modern relational DBMSs:

- Row-stores
- Column-stores
Row-stores

a.k.a. row-wise storage or \( n \)-ary storage model, NSM:
Column-stores

a.k.a. column-wise storage or decomposition storage model, DSM:
The effect on query processing

Consider, e.g., a selection query:

```sql
SELECT COUNT(*)
FROM lineitem
WHERE l_shipdate = "2009-09-26"
```

This query typically involves a **full table scan**.
A full table scan in a row-store

In a row-store, all **rows** of a table are stored sequentially on a database page.
A full table scan in a row-store

In a row-store, all **rows** of a table are stored sequentially on a database page.

```
+-----+-----+-----+-----+-----+
|     |     |     |     |     |
| l_shipdate | tuple |
|     |     |     |     |     |
|     |     |     |     |     |
|     |     |     |     |     |
|     |     |     |     |     |
|     |     |     |     |     |
|     |     |     |     |     |
|     |     |     |     |     |
|     |     |     |     |     |
|     |     |     |     |     |
+-----+-----+-----+-----+-----+
```
A full table scan in a row-store

In a row-store, all **rows** of a table are stored sequentially on a database page.

With every access to a `l_shipdate` field, we load a large amount of **irrelevant** information into the cache.
A "full table scan" on a column-store

In a column-store, all values of one column are stored sequentially on a database page.

l_shipdate(s)
A "full table scan" on a column-store

In a column-store, all values of one column are stored sequentially on a database page.

\texttt{l\_shipdate(s)}

All data loaded into caches by a "\texttt{l\_shipdate scan}" is now actually relevant for the query.
Column-store advantages

• All data loaded into caches by a “l_shipdate scan” is now actually relevant for the query.
  → Less data has to be fetched from memory.
  → Amortize cost for fetch over more tuples.
  → If we’re really lucky, the full (l_shipdate) data might now even fit into caches.

• The same arguments hold, by the way, also for disk-based systems.

• Additional benefit: Data compression might work better.
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
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

$\sim$ Processing of compressed values is not straight forward
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

```
... State ...
Bavaria
Hesse
Hesse
Saxony
Sx. Anhalt
Sx. Anhalt
Thuringia
Thuringia
Thuringia
...
```

```
# State
1  Bavaria
2  Hesse
1  Saxony
2  Sx. Anhalt
3  Thuringia
...
```

Run Length Encoding (RLE)

```
# State
1  0000
2  0001
1  0010
2  0011
3  0100
...
```

Dictionary Encoding and RLE
**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

### Example Table

<table>
<thead>
<tr>
<th>State</th>
<th>Common Value</th>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thuringia</td>
<td>1</td>
<td>0100</td>
</tr>
<tr>
<td>NULL</td>
<td>0</td>
<td>0010</td>
</tr>
<tr>
<td>Saxony</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>NULL</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>Hesse</td>
<td>1</td>
<td>0011</td>
</tr>
<tr>
<td>Bavaria</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>NULL</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>Sx. Anhalt</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>NULL</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<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>
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>...</th>
<th>Customer Status</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Platinum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Silver</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Platinum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gold</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gold</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Silver</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Platinum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gold</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Silver</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>...</th>
<th>Platinum</th>
<th>Gold</th>
<th>Silver</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<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>...</td>
</tr>
<tr>
<td>39104</td>
<td></td>
</tr>
<tr>
<td>39106</td>
<td></td>
</tr>
<tr>
<td>39108</td>
<td></td>
</tr>
<tr>
<td>39130</td>
<td></td>
</tr>
<tr>
<td>80336</td>
<td></td>
</tr>
<tr>
<td>80339</td>
<td></td>
</tr>
<tr>
<td>80807</td>
<td></td>
</tr>
<tr>
<td>80809</td>
<td></td>
</tr>
<tr>
<td>80933</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Country</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td>1</td>
</tr>
<tr>
<td>USA</td>
<td>01</td>
</tr>
<tr>
<td>Germany</td>
<td>001</td>
</tr>
<tr>
<td>Russia</td>
<td>0001</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
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)

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)
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

[Diagram illustrating frequency partitioning]

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>MonetDB</th>
<th>SAP HANA</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>2,662.777</td>
<td>64.599</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>refid</td>
<td>4,122.000</td>
<td>4,495.983</td>
<td>109.073</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>insert_offset</td>
<td>4,122.000</td>
<td>0.076</td>
<td>0.002</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>base</td>
<td>1,030.635</td>
<td>0.771</td>
<td>0.075</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>base_call_quality</td>
<td>4,122.000</td>
<td>1.758</td>
<td>0.043</td>
</tr>
<tr>
<td>6</td>
<td>grch37</td>
<td>id</td>
<td>950.875</td>
<td>2,614.746</td>
<td>274.983</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>position</td>
<td>950.875</td>
<td>1,782.782</td>
<td>187.489</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>base</td>
<td>237.875</td>
<td>0.003</td>
<td>0.001</td>
</tr>
<tr>
<td>9</td>
<td>reads</td>
<td>id</td>
<td>43.125</td>
<td>107.552</td>
<td>249.396</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>mapping_quality</td>
<td>43.125</td>
<td>0.010</td>
<td>0.024</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>overall</td>
<td>19,744.510</td>
<td>11,666.458</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>
## 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>overall</td>
<td></td>
<td>19,744.510</td>
<td>11,666.458</td>
</tr>
</tbody>
</table>
Column-store trade-offs

Tuple recombination can cause considerable cost.

- Need to perform many joins.
- Workload-dependent trade-off.

Source: [Copeland and Khoshafian, 1985]
An example: Binary Association Tables in MonetDB

MonetDB makes this explicit in its data model.

- **All** tables in MonetDB have two columns (“head” and “tail”).

<table>
<thead>
<tr>
<th>NAME</th>
<th>AGE</th>
<th>SEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>34</td>
<td>m</td>
</tr>
<tr>
<td>Angelina</td>
<td>31</td>
<td>f</td>
</tr>
<tr>
<td>Scott</td>
<td>35</td>
<td>m</td>
</tr>
<tr>
<td>Nancy</td>
<td>33</td>
<td>f</td>
</tr>
</tbody>
</table>

• Each column yields one binary association table (BAT).
• Object identifiers (oids) identify matching entries (BUNs).
• Often, oids can be implemented as virtual oids (voids).
→ Not explicitly materialized in memory.
An example: Binary Association Tables in MonetDB

MonetDB makes this explicit in its data model.

- **All** tables in MonetDB have two columns ("head" and "tail").

<table>
<thead>
<tr>
<th>oid</th>
<th>NAME</th>
<th>AGE</th>
<th>SEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₁</td>
<td>John</td>
<td>34</td>
<td>m</td>
</tr>
<tr>
<td>o₂</td>
<td>Angelina</td>
<td>31</td>
<td>f</td>
</tr>
<tr>
<td>o₃</td>
<td>Scott</td>
<td>35</td>
<td>m</td>
</tr>
<tr>
<td>o₄</td>
<td>Nancy</td>
<td>33</td>
<td>f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>oid</th>
<th>NAME</th>
<th>AGE</th>
<th>SEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₁</td>
<td>John</td>
<td>34</td>
<td>m</td>
</tr>
<tr>
<td>o₂</td>
<td>Angelina</td>
<td>31</td>
<td>f</td>
</tr>
<tr>
<td>o₃</td>
<td>Scott</td>
<td>35</td>
<td>m</td>
</tr>
<tr>
<td>o₄</td>
<td>Nancy</td>
<td>33</td>
<td>f</td>
</tr>
</tbody>
</table>

- Each column yields one **binary association table (BAT)**.
- **Object identifiers** (oids) identify matching entries (BUNs).
An example: Binary Association Tables in MonetDB

MonetDB makes this explicit in its data model.

- **All** tables in MonetDB have two columns ("head" and "tail").

<table>
<thead>
<tr>
<th>oid</th>
<th>NAME</th>
<th>AGE</th>
<th>SEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₁</td>
<td>John</td>
<td>34</td>
<td>m</td>
</tr>
<tr>
<td>o₂</td>
<td>Angelina</td>
<td>31</td>
<td>f</td>
</tr>
<tr>
<td>o₃</td>
<td>Scott</td>
<td>35</td>
<td>m</td>
</tr>
<tr>
<td>o₄</td>
<td>Nancy</td>
<td>33</td>
<td>f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>oid</th>
<th>NAME</th>
<th>AGE</th>
<th>SEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₁</td>
<td>John</td>
<td>34</td>
<td>m</td>
</tr>
<tr>
<td>o₂</td>
<td>Angelina</td>
<td>31</td>
<td>f</td>
</tr>
<tr>
<td>o₃</td>
<td>Scott</td>
<td>35</td>
<td>m</td>
</tr>
<tr>
<td>o₄</td>
<td>Nancy</td>
<td>33</td>
<td>f</td>
</tr>
</tbody>
</table>

- Each column yields one **binary association table (BAT)**.
- **Object identifiers** (oids) identify matching entries (BUNs).
- Often, oids can be implemented as **virtual oids (voids)**.
  → Not explicitly materialized in memory.
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

Picture taken from [Abadi et al., 2007]
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

<table>
<thead>
<tr>
<th>custkey</th>
<th>region</th>
<th>nation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Asia</td>
<td>China</td>
</tr>
<tr>
<td>2</td>
<td>Europe</td>
<td>France</td>
</tr>
<tr>
<td>3</td>
<td>Asia</td>
<td>India</td>
</tr>
</tbody>
</table>

Hash table with keys 1 and 3

Apply region = 'Asia' on Supplier table

<table>
<thead>
<tr>
<th>suppkey</th>
<th>region</th>
<th>nation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Asia</td>
<td>Russia</td>
</tr>
<tr>
<td>2</td>
<td>Europe</td>
<td>Spain</td>
</tr>
</tbody>
</table>

Hash table with key 1

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 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 1 0 1 1 0 1 1

matching fact table bitmap for cust. dim. join

= 1 1 1 1 0 0 0 0 0

Bitwise And

fact table tuples that satisfy all join predicates

Hash table with key 1

1 0 1 1 0 0 0 0 0

Probe

Hash table with keys 01011997, 01021997, and 01031997

1 1 1 1 1 1 1 1
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]

Picture taken from [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]
Conclusion

• **Row**-stores store complete tuples sequentially on a database page

• **Column**-stores store all values of one column sequentially on a database page

• Depending on the workload column-stores or row-stores are more advantageous
  
  • Tuple reconstruction is overhead in column-stores
  • Analytical workloads that process few columns at a time benefit from column-stores

→ One data storage approach is not optimal to serve all possible workloads
Take home messages

• **Modern hardware** offers performance improvements for DB applications → disk vs. main-memory access speed

• **Rethinking the architecture of DBMSs** to adapt them on changes in hardware pays off → single- vs. multi-threaded OLTP engines

• **New DBMS architectures** mean that we have to
  • **solve old problems** again → durability and availability
  • **optimize for different things** → cache effectiveness
Invitation

Your are invited to join our research on main-memory databases and databases on new hardware, e.g., in form of:

• Thesis
  www.dbse.ovgu.de/Thesis_Jobs.html

• Scientific Team Project: Modern Database Technologies
  http://www.dbse.ovgu.de/Lehre/Scientific+Team+Project.html

• Requirements:
  • Motivation
  • (Programming skills)
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.

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

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

