In-Memory
Data Structures and Databases

Jens Krueger

Enterprise Platform and Integration Concepts
Hasso Plattner Institute
What to take home from this talk?

Answer to the following questions:

■ What makes an in-memory database fast?
■ What are differences of an in-memory database to disk-based systems?
■ How does the physical data representation affect the performance of an in-memory database?
■ How to leverage sequential data access?
■ How can compression improve read access?
Recap

Jens Krueger

Enterprise Platform and Integration Concepts
Hasso Plattner Institute
## Recap: Workload Characteristics

<table>
<thead>
<tr>
<th>OLTP</th>
<th>OLAP/DSS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full row operations</strong></td>
<td>Retrieve small number of columns</td>
</tr>
<tr>
<td><strong>Simple Queries</strong></td>
<td>Complex Queries</td>
</tr>
<tr>
<td><strong>Detail Row Retrieval</strong></td>
<td>Aggregation and Group By</td>
</tr>
<tr>
<td><strong>Inserts/Updates/Selects</strong></td>
<td>Mainly Selects</td>
</tr>
<tr>
<td><strong>Short Transactions</strong></td>
<td>Long Transactions</td>
</tr>
<tr>
<td><strong>Small Found Sets</strong></td>
<td>Large Found Sets</td>
</tr>
<tr>
<td><strong>Pre-determined Queries</strong></td>
<td>Adhoc Queries</td>
</tr>
<tr>
<td><strong>Real Time Updates</strong></td>
<td>Batch Updates</td>
</tr>
<tr>
<td>„Source of Truth“</td>
<td>Alternative representation</td>
</tr>
</tbody>
</table>

Clark D. French, „Teaching an OLTP Database Kernel Advanced Datawarehousing Techniques“ ICDE 97
Recap: Trends in Enterprise Apps

Today's Enterprise Applications
- Complex processes
- Increased data set (but real-world events driven)
- Separated into OLTP and OLAP

Enterprise data management
- Wide schemas
- Sparse data with limited domain
- Workload consists of complex, analytic-style queries
- Workload is mostly:
  - Set processing
  - Read access
  - Insert instead of updates

Mixed Workload
Why is an in-memory database faster than a fully cached disk-based database?
Excursus: Disk-based Databases

Jens Krueger

Enterprise Platform and Integration Concepts
Hasso Plattner Institute
Excursus: Magnetic Disks

- Random Access (even though slow)
- Inexpensive
- Non-volatile

Parts of a magnetic disk
- Platter: covered with magnetic recording material (turning)
- Track: logical division of platter surface
- Sector: hardware division of tracks
- Block: OS division of tracks
  Typical block sizes: 512B, 2KB, 4KB
- Read/write head (moving)
Files on Disk

- Metadata defines
  - Tables
  - Attributes
  - Data Types

- Stored are (data)
  - Logs
  - Records (== tuple)
  - Indices

- Data is stored in files
  - A file has one or more pages
  - A page contains one or more records.
Rows, Columns, and the Page Layout

- **Row-oriented page layout** (n-ary storage model)

- **Column-oriented page layout** (decomposed storage model)
Buffer Management

- **Buffer** caches copies of pages in main memory
- **Buffer Manager** *maintains* these pages
  - **Hit:** requested page in buffer
  - **Miss:** page on disk
    - Allocate page frame
    - Read page
  - **Page replacement**
    - Dirty flag for write back
    - Least Recently Used (LRU)
    - Most Recently Used (MRU)

![Diagram showing page request, buffer, page request, disk, and page replacement logic.](image-url)
In a Nutshell

- Optimizations
  - Sequential Access
  - Buffering and scheduling algorithms
  - In-memory indices to pages
  - Pre-calculation and materialization
  - Etc.

- Page structure leads to
  - Good write performance
  - Efficient single tuple access
  - **Overhead** if single attributes scanned
    - regardless of disk throughput -
Why is an in-memory database faster than a fully cached disk-based database?

- Disk access
  - Low throughput
  - Slow random access

- Buffer Management

- Disk-oriented data structures
  (even in main memory)
  - Page layout
  - Indices
Question

Does this mean to keep data in main memory to achieve performance while the physical data representation can be neglected?

Why?
Memory Access

Jens Krueger

Enterprise Platform and Integration Concepts
Hasso Plattner Intitute
Memory hierarchy:
- Capacity restricted by price/performance
- SRAM vs. DRAM (refreshing needed every 64ms)
- SRAM is very fast but very expensive

Memory is organized in hierarchies
- Fast but small memory on the top
- Slow but lots of memory at the bottom

<table>
<thead>
<tr>
<th>technology</th>
<th>latency</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td></td>
<td>bytes</td>
</tr>
<tr>
<td>L1 Cache</td>
<td></td>
<td>KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td></td>
<td>MB</td>
</tr>
<tr>
<td>Main Memory</td>
<td></td>
<td>GB</td>
</tr>
<tr>
<td>SRAM</td>
<td>&lt; 1 ns</td>
<td></td>
</tr>
<tr>
<td>L1 Cache</td>
<td>~ 1 ns</td>
<td></td>
</tr>
<tr>
<td>L2 Cache</td>
<td>&lt; 10 ns</td>
<td></td>
</tr>
<tr>
<td>Main Memory</td>
<td>100 ns</td>
<td></td>
</tr>
</tbody>
</table>
Capacity vs. Speed (latency)

- **CPU**
  - L1 Cache
  - L2 Cache
  - Main Memory
- **Magnetic Disk**

<table>
<thead>
<tr>
<th>latency</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 1 ns</td>
<td>bytes</td>
</tr>
<tr>
<td>~ 1 ns</td>
<td>KB</td>
</tr>
<tr>
<td>&lt; 10 ns</td>
<td>MB</td>
</tr>
<tr>
<td>100 ns</td>
<td>GB</td>
</tr>
<tr>
<td>(~ 10 000 000 ns)</td>
<td>TB</td>
</tr>
</tbody>
</table>
In DBMS, on disk as well as in memory, data processing is often:

- Not CPU bound
- **But** bandwidth bound
- “I/O Bottleneck”

CPU could process data faster

**Memory Access:**

- **Not** truly random (in the sense of constant latency)
- Data is read in **blocks**/cache lines
- Even if only parts of a block are requested

Potential **waste** of bandwidth
Memory Basics I

- **Cache**
  Small but fast memory, which keeps data from main memory for fast access.

  **Cache performance is crucial**
  - Similar to disk cache (e.g. buffer pool)

  **But:** Caches are controlled by hardware.

- **Cache hit**
  Data was found in the cache.
  Fastest data access since no lower level is involved.

- **Cache miss**
  Data was **not** found in the cache. CPU has to load data from main memory into cache (**miss penalty**).
Memory Basics II

- **Cache lines**
  The cache is partitioned into lines.
  - Data is read or written as whole line
  - Size: 4-64 bytes

→ Due to unnecessary data in cache lines the cache gets **polluted**.
Locality is King!

To improve cache behavior
- Increase cache capacity
- Exploit locality
  - Spatial: related data is close (nearby references are likely)
  - Temporal: Re-use of data (repeat reference is likely)

To improve locality
- Non random access (e.g. scan, index traversal):
  - Leverage sequential access patterns
  - Clustering data to a cache lines
  - Partition to avoid cache line pollution (e.g. vertical decomposition)
  - Squeeze more operations into a cache line

- Random access (hash join):
  - Partition to fit in cache
A Simple C++

- Logical

- Physical

```c
int *table = (int*) calloc((rows * columns), sizeof(int));
```

...
Example for Sequential Access

```c
for (r = 0; r < rows; r++)
    for (c = 0; c < columns; c++)
        sum += table[r * columns + c];
```

**Simulates sequential access**

- All data in a cache line is read
- Prefetching and pipelining further **improve** performance
Example for Traversal Sequential Access

```cpp
for (c = 0; c < columns; c++)
    for (r = 0; r < rows; r++)
        sum += table[c * columns + r];
```

**Simulates traversal sequential access**

- Fixed stride (access offset) leads to cache misses
- Cache size / performance can be measured by varying the stride
A Simple C++

- Logical

- Physical

```cpp
int *table = (int*) calloc((rows * columns), sizeof(int));
```

...
```cpp
#include <sys/time.h>
#include <vector>
#include <iostream>
using namespace std;

#define C_NUMRUNS 1

typedef unsigned int uint;

void seq_read(unsigned int rows, unsigned int columns) {
    struct timeval start1, end1, start2, end2;
    long time;
    unsigned int r, c, table_size;
    int w;
    unsigned int seq_sum, seq2_sum, stride_sum;

    cout << "Fill table" << endl;
    int* table = (int*) calloc((rows * columns), sizeof(int));
    int* read = (int*) malloc(columns * sizeof(int));

    for (r = 0; r < rows; r++)
        for (c = 0; c < columns; c++)
            table[r * columns + c] = (unsigned int) random() % 99999999;
    table_size = (((rows * columns) * sizeof(int)) / 1024 / 1024);
    cout << "Table: " << table_size << "MB" << endl;

    cin >> w;

    cout << "Sequential Access " << endl;
    seq_sum = 0; time = 0;
    gettimeofday(&start1, NULL);
    for (r = 0; r < rows; r++)
        for (c = 0; c < columns; c++)
            read[c] = table[r * columns + c];
    gettimeofday(&end1, NULL);
    time = (end1.tv_sec - start1.tv_sec) * 1000000 + (end1.tv_usec - start1.tv_usec);
    cout << "Sum:  " << seq_sum << endl;
    cout << "Time:  " << time << " usec" << (time / 1000.0) << " msec" << (table_size / (time / 1000.0 / 1000.0)) << "MB/s" << endl;

    free(table);
    free(read);
}

int main(int argc, char* argv[]) {
    unsigned int rows = 3000000;
    unsigned int columns = 300;
    seq_read(rows, columns);
    cout << "######### Finish" << endl;
    return 0;
}
```
In-Memory Databases

Jens Krueger

Enterprise Platform and Integration Concepts
Hasso Plattner Institute
In-Memory Database

In an In-Memory Database (IMDB)

- Data resides **permanently** in main memory
- Main Memory is the **primary** "persistence"
- Still: logging to **disk**/recovery from **disk**
- Main memory access is the new **bottleneck**
- Cache-conscious algorithms/data structures are **crucial** (locality is king)

**Today's Main Memory Technology**

- Increased size: up to 2 TB of main memory on one main board as of today
- Increased bandwidth: up 30GB/s
- Latency is hidden by caches (memory hierarchy)
In an In-Memory Database (IMDB)

- Data resides permanently in main memory
- Main Memory is the primary "persistence"
- Still: logging to disk/recovery from disk
- Main memory access is the new bottleneck
- Cache-conscious algorithms/data structures are crucial (locality is king)

Differences to disk-based systems

- Volatile
- Direct access
- Access time
- Access cost
Question

Does an entire database fit in main memory?
Question + Answer

Does an entire database fit in main memory?

■ Yes:
  □ Limited DB size, i.e. enterprise applications
  □ Due to data compression (factor 10 feasible)
  □ Redundant-free data schemas

■ No:
  □ Data could be partitioned over nodes
  □ Data aging strategies for extended memory hierarchies (e.g. SSD/disks for non-active data)
More Main Memory for Disk-based DBMS?

**What is the difference between an IMDB and a disk-based DB with a large cache?**

- Different optimizations for data structures, e.g.
  - Page layout
  - No access through a buffer manager
  - Index structures
  - Cache-aware data organization
  - Random access capabilities, e.g. for locking

- As disk-based DB’s can have in-memory optimization, they still would have to maintain different data structures.
The physical data layout with regards to the workload has a significant influence on the cache behavior of the IMDB.

- Tuples are spanned over cache lines
- Wrong layout can lead to lots of (expensive) cache misses
- Row- or column-oriented can reduce cache misses if matching workload is applied
How to optimize an IMDB?
How to optimize an IMDB?
- Exploit sequential access
- Leverage locality
Row- or Column-oriented Storage

SELECT *  
FROM Sales Orders  
WHERE Document Number = '95779216'

SELECT SUM(Order Value)  
FROM Sales Orders  
WHERE Document Date > 2009-01-20
Row-oriented storage

<table>
<thead>
<tr>
<th>A1</th>
<th>B1</th>
<th>C1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
<tr>
<td>A3</td>
<td>B3</td>
<td>C3</td>
</tr>
<tr>
<td>A4</td>
<td>B4</td>
<td>C4</td>
</tr>
</tbody>
</table>
Row-oriented storage
Row-oriented storage
Row-oriented storage

<table>
<thead>
<tr>
<th>A1</th>
<th>B1</th>
<th>C1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
<tr>
<td>A3</td>
<td>B3</td>
<td>C3</td>
</tr>
<tr>
<td>A4</td>
<td>B4</td>
<td>C4</td>
</tr>
</tbody>
</table>
Row-oriented storage
Column-oriented storage

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
<tr>
<td>A3</td>
<td>B3</td>
<td>C3</td>
</tr>
<tr>
<td>A4</td>
<td>B4</td>
<td>C4</td>
</tr>
</tbody>
</table>
Column-oriented storage
Column-oriented storage

<table>
<thead>
<tr>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Column-oriented storage

<table>
<thead>
<tr>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
</table>
struct Tuple {
    int a, b, c;
};

Tuple data[4];
fill(data);

Tuple third = data[3];
Example: OLTP-Style Query

```c
struct Tuple {
    int a, b, c;
};

Tuple data[4];
fill(data);

Tuple third = data[3];
```
Example: OLAP-Style Query

```c
struct Tuple {
int a, b, c;
};

Tuple data[4];
fill(data);

int sum = 0;

for(int i = 0; i<4; i++)
sum += data[i].a;
```
Example: OLAP-Style Query

```c
struct Tuple {
    int a, b, c;
};

Tuple data[4];
fill(data);

int sum = 0;

for(int i = 0; i < 4; i++)
    sum += data[i].a;
```
Mixed Workloads

- Mixed Workloads involve attribute- and entity-focused queries

**OLTP-style queries**

<table>
<thead>
<tr>
<th>A1</th>
<th>B1</th>
<th>C1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
<tr>
<td>A3</td>
<td>B3</td>
<td>C3</td>
</tr>
<tr>
<td>A4</td>
<td>B4</td>
<td>C4</td>
</tr>
</tbody>
</table>

**OLAP-style queries**

<table>
<thead>
<tr>
<th>A1</th>
<th>B1</th>
<th>C1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
<tr>
<td>A3</td>
<td>B3</td>
<td>C3</td>
</tr>
<tr>
<td>A4</td>
<td>B4</td>
<td>C4</td>
</tr>
</tbody>
</table>
Mixed Workloads: Choosing the Layout

<table>
<thead>
<tr>
<th>Layout</th>
<th>OLTP-Misses</th>
<th>OLAP-Misses</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Column</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

**OLTP-style queries**

- A1
- A2
- A3
- A4
- B1
- B2
- B3
- B4
- C1
- C2
- C3
- C4

**OLAP-style queries**

- A1
- A2
- A3
- A4
- B1
- B2
- B3
- B4
- C1
- C2
- C3
- C4
Question

What is the best layout for mixed workloads?
Hybrid: Grouping of Columns

<table>
<thead>
<tr>
<th></th>
<th>A1</th>
<th>B1</th>
<th>C1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td></td>
<td>B2</td>
<td>C2</td>
</tr>
<tr>
<td>A3</td>
<td></td>
<td>B3</td>
<td>C3</td>
</tr>
<tr>
<td>A4</td>
<td></td>
<td>B4</td>
<td>C4</td>
</tr>
</tbody>
</table>
Hybrid: Grouping of Columns
Hybrid: Grouping of Columns

A1 A2 A3 A4 B1 C1

B2 C2
B3 C3
B4 C4
Hybrid: Grouping of Columns

<table>
<thead>
<tr>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
<th>B1</th>
<th>C1</th>
<th>B2</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hybrid: Grouping of Columns

A1 A2 A3 A4 B1 C1 B2 C2 B3 C3
Hybrid: Grouping of Columns

| A1 | A2 | A3 | A4 | B1 | C1 | B2 | C2 | B3 | C3 | B4 | C4 |
Hybrid: Grouping of Columns

Access tuple 3

Query attribute A

<table>
<thead>
<tr>
<th>Layout</th>
<th>OLTP-Misses</th>
<th>OLAP-Misses</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Column</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Hybrid</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
What other optimization for an IMDB?
Compression in In-Memory Databases

Jens Krueger

Enterprise Platform and Integration Concepts
Hasso Plattner Intitute
Motivation

- Main memory is the new bottleneck
- Processor speed increases faster than memory speed
- Trade CPU time to compress and decompress data
- Compression
  - **Reduces** I/O operations to main memory
  - Leads to **less** cache misses due to more information on a cache line
  - Enables operations **directly** on compressed data
  - Allows to **offset** by the use of fixed-length data types
Compression Techniques

- Lightweight compression techniques:
  - **Lossless**
    - Reduce the amount of data
    - Improve query execution
    - Better utilizes cache lines
  - Techniques
    - Run Length Encoding
    - Null Suppression
    - Bit Vector Encoding
    - Dictionary Encoding
Run Length Encoding (RLE)

- Subsequent equal values are stored as one value with offset (value, run_length)
- Especially useful for sorted columns
- But:
  - If column store works with TupleId, only sorting by one column is possible
Null Suppression

- Remove leading 0’s
- Most effective when encoding random sequence of small integers
  - int x = 7; uses 32 bits but first 29 are 0’s
  - store (length, encoding) => (3, 111)
- Optimization: store byte count for next 4 values as two bits in one byte
Bit vector encoding

- Store a bitmap for each distinct value
- Values to encode: a b a a c c b
  - a => (1 0 1 1 0 0 0)
  - b => (0 1 0 0 0 0 1)
  - c => (0 0 0 0 1 1 0)
- Useful with few distinct values
Dictionary Encoding

- Store distinct values once in separate mapping table (the dictionary)
- Associate unique mapping key for each distinct value
- Store mapping key instead of value in value table
Example (1)

- Store fixed length strings of 32 characters
  - SQL-Speak: CHAR(32) - 32 Bytes
  - 1 Million entries consume 32 * 10^6 Bytes
  - ~ 32 Megabytes
Example (2)

- Associate 4 byte valueID with distinct value
- Dictionary: assume 200,000 distinct values
  - Each: 1 key, 1 value => 36 Bytes
  - ~ 7.2 Megabytes
  - 1 million * 4 Bytes = ~ 4 Megabytes
- Overall: 11.2 Megabytes
- 64 byte cache line
  - Uncompressed: 2 values per cache line
  - Compressed: 16 valueID’s per cache line
Question

How can this compression technique further be improved?

With regards to:

- **Amount** of data
- Query **execution**
Answer

- Amount of data
  - Idea: compress valueID’s
  - Use only bits needed to represent the cardinality of distinct values - $\log_2(\text{distinct values})$
  - Optimal for only a few distinct values
  - Re-encoding if more bits to encode needed

- Query execution
  - Use order-preserving dictionaries
  - ValueID’s have same order as uncompressed values
  - $\text{value1} < \text{value2} \iff \text{valueID1} < \text{valueID2}$
Materialization in Column Stores

Jens Krueger

Enterprise Platform and Integration Concepts
Hasso Plattner Institute
Strategies for Tuple Reconstruction

Strategies:

- **Early** materialization
  Create a row-wise data representation at the first operator

- **Late** materialization
  Operate on columns as long as possible

Reference: D. Abadi: SIGMOD 2009
Example:

**Query:**

```
SELECT kunnr, sum(dmbtr) 
FROM BSEG 
WHERE gjahr = 4 
AND bukrs = 1 
GROUP BY kunnr
```

**Table BSEG**

<table>
<thead>
<tr>
<th>gjahr</th>
<th>bukrs</th>
<th>kunnr</th>
<th>dmbtr</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>80</td>
</tr>
</tbody>
</table>

Reference: D. Abadi: SIGMOD 2009
Early materialization

Query:
SELECT kunnr, sum(dmbtr) FROM BSEG WHERE gjahr = 4 AND bukrs = 1 GROUP BY kunnr

- Create rows first
  - Need to construct ALL tuples
  - Need to decompress data
  - Poor memory bandwidth utilization

Reference: D. Abadi: SIGMOD 2009
Late materialization I

- Operate on columns

**Query:**
```
SELECT kunnr, sum(dmbtr)
FROM BSEG
WHERE gjahr = 4
AND bukrs = 1
GROUP BY kunnr
```

Reference: D. Abadi: SIGMOD 2009
Late materialization II

- Operate on columns

Query:
SELECT kunnr, sum(dmbtr)
FROM BSEG
WHERE gjahr = 4
AND bukrs = 1
GROUP BY kunnr

Reference: D. Abadi: SIGMOD 2009
Late materialization III

- Operate on columns

```
Query:
SELECT kunnr, sum(dmbtr)
FROM BSEG
WHERE gjahr = 4
AND bukrs = 1
GROUP BY kunnr
```

Reference: D. Abadi: SIGMOD 2009
Late materialization IV

- Operate on columns

```
Query:
SELECT kunnr, sum(dmbtr)
FROM BSEG
WHERE gjahr = 4
AND bukrs = 1
GROUP BY kunnr
```

Reference: D. Abadi: SIGMOD 2009