Data Structures for In-Memory Databases

Jens Krueger

Enterprise Platform and Integration Concepts Group

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 a in-memory database?
- How to leverage sequential data access?
- How can compression improve read access?

Recap

Jens Krueger

Enterprise Platform and Integration Concepts Group

Recap: Workload Characteristics

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

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**

Question

Why is an in-memory database faster than a fully cached disk-based database?

Excursus: Disk-based Databases

Jens Krueger

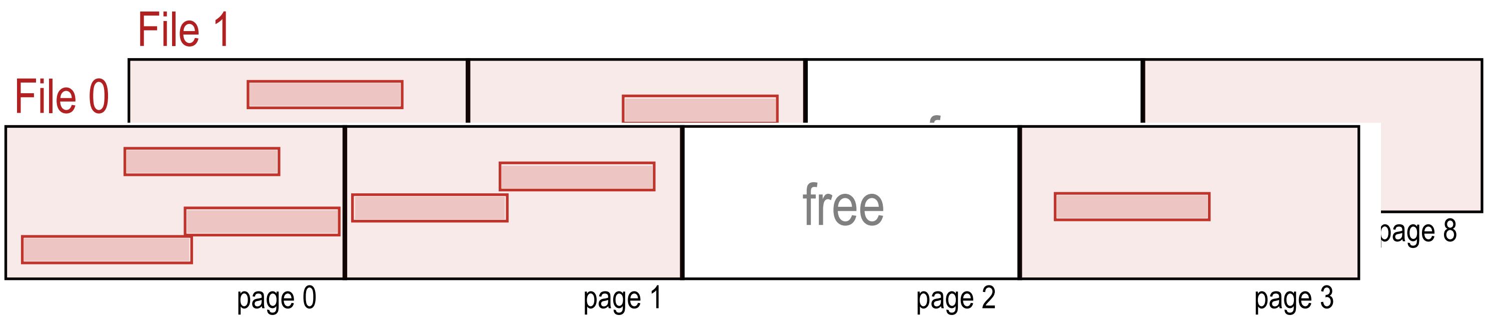
Enterprise Platform and Integration Concepts Group

Excursus: Magnetic Disks

- Random Access (even though slow)
- Inexpensive
- Non-volatile
- Parts of an 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

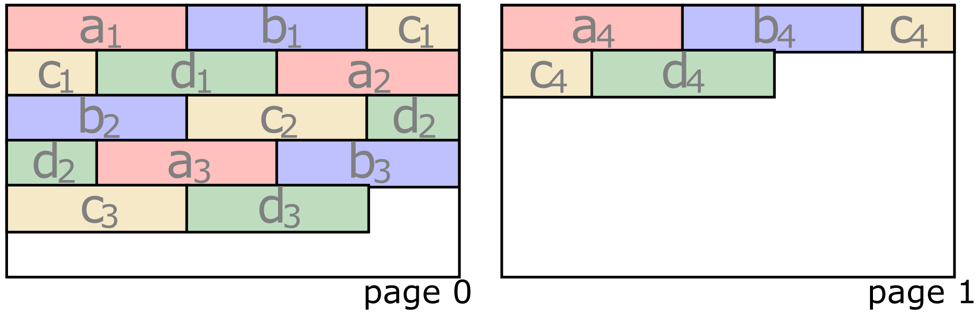
- Metadata defines
  - Tables
  - Attributes
  - Data Types
- Stored are (data)
  - Logs
  - Records (== tuple)
  - Indices
- Data is stored in files
  - A file has one or more pages
  - A page contains of one or more records.



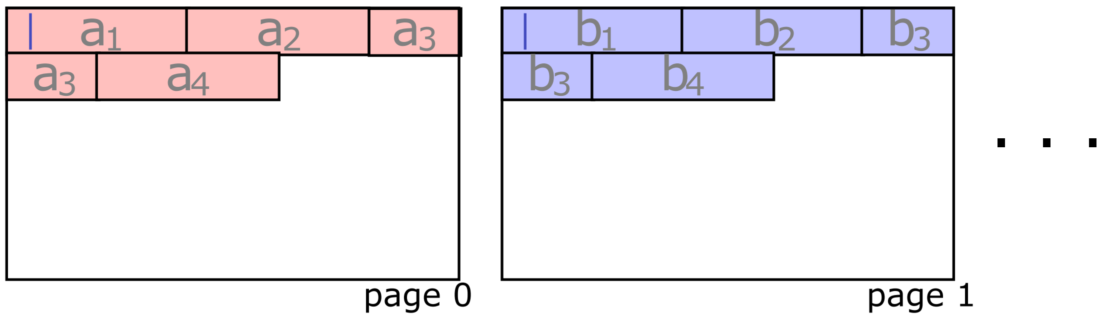
Rows, Columns, and the Page Layout

a_1	b_1	c_1	d_1
a_2	b_2	c_2	d_2
a_3	b_3	c_3	d_3
a_4	b_4	c_4	d_4

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

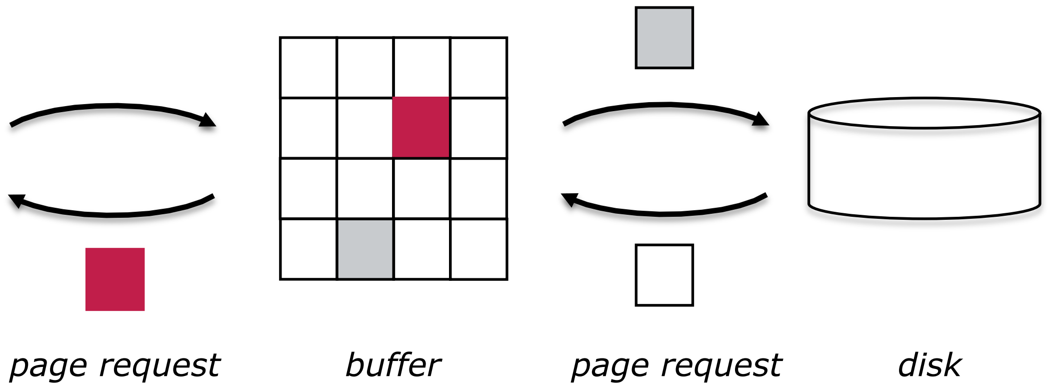


- **Column-oriented page layout** (decomposed storage model)



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)



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 -

Question + Answer

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 Group

Capacity vs. Speed (latency)

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

	technology	latency	size
CPU	SRAM	< 1 ns	bytes
L1 Cache	SRAM	~ 1 ns	KB
L2 Cache	SRAM	< 10 ns	MB
Main Memory	DRAM	100 ns	GB

Capacity vs. Speed (latency)

	latency	**size**
CPU	< 1 ns	bytes
L1 Cache	~ 1 ns	KB
L2 Cache	< 10 ns	MB
Main Memory	100 ns	GB
Magnetic Disk	**~ 10 000 000 ns** (10 ms)	**TB**

Data Processing

In DBMS, on disk as well as in memory, data processing is often:

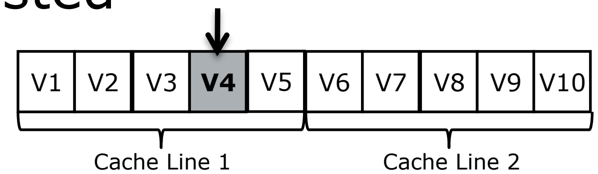
- Not CPU bound
- **But** bandwidth bound
- “I/O Bottleneck”

➡ 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

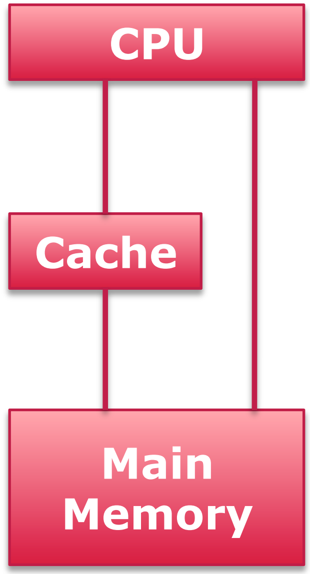
➡ Potential **waste** of bandwidth



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.

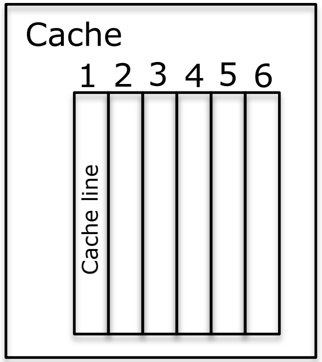
  Fastest data access since no lower level is involved.

- **Cache miss**

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

- **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

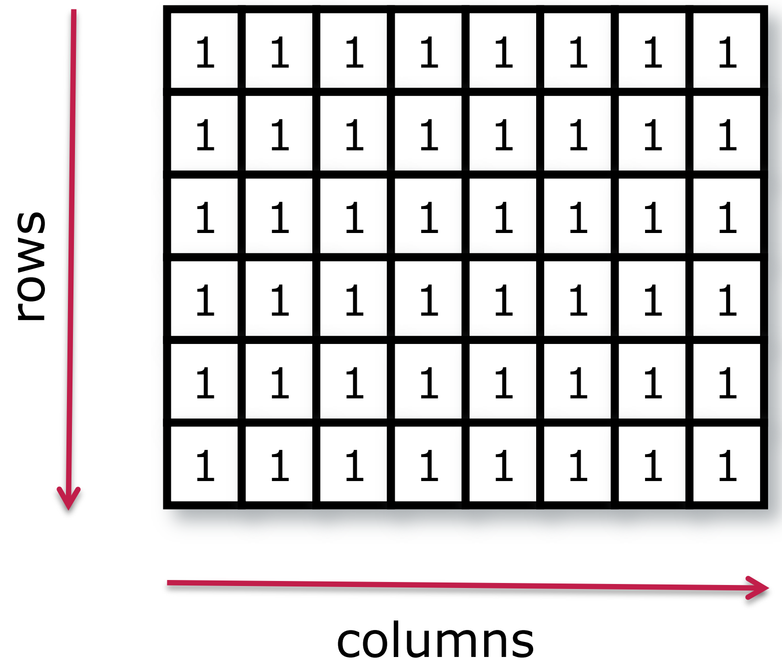
- Increase cache capacity
- Exploit locality
  - Spatial: related data is close (nearby references are likely)
  - Temporal: Re-use of data (repeat reference is likely)

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

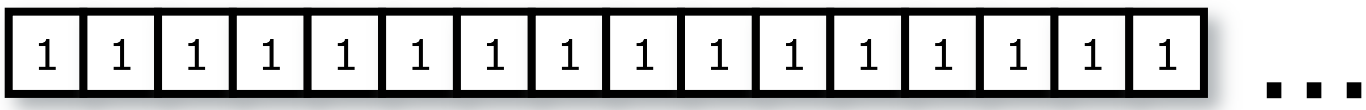
A Simple C++

- Logical



- Physical

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



Example for Sequential Access

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

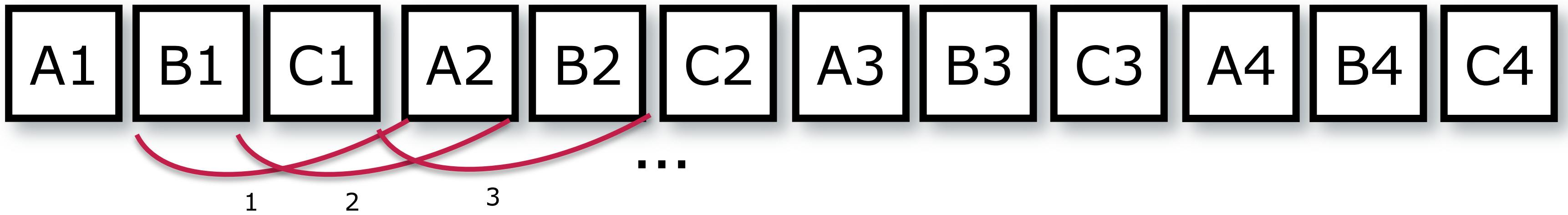
1 2 3 4 ...

```
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



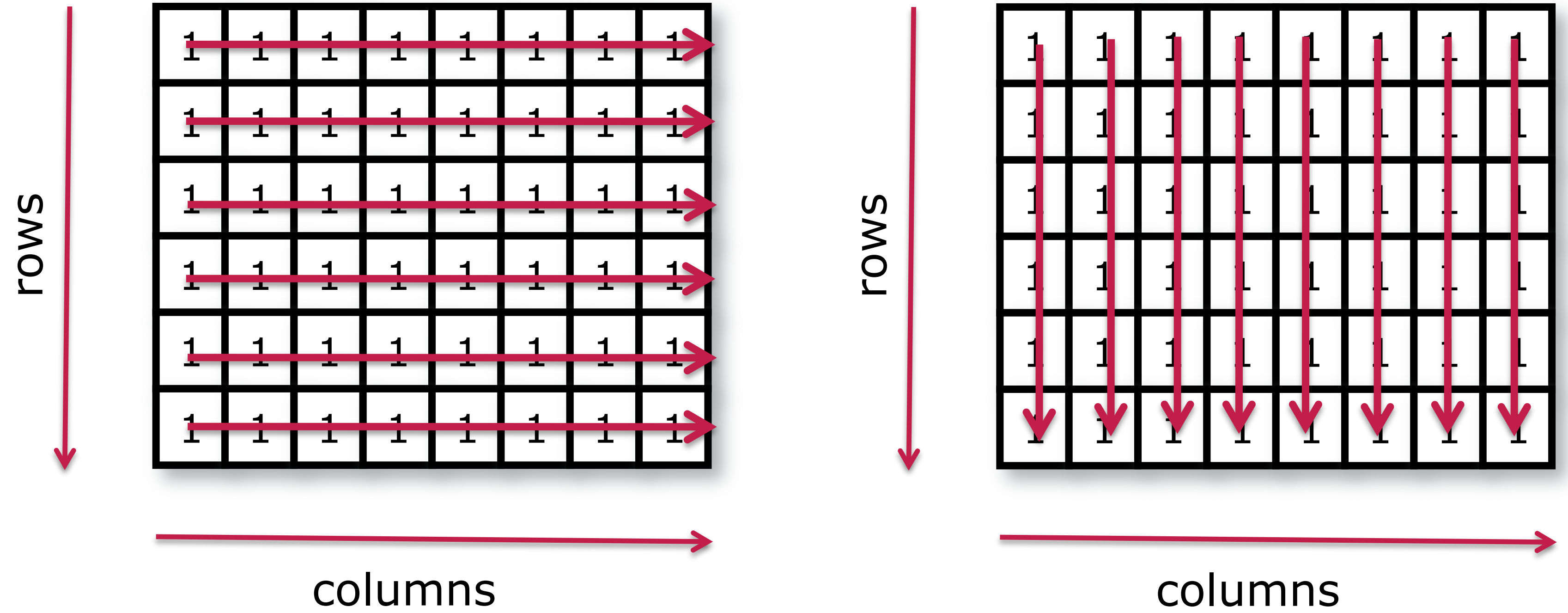

```
for (c = 0; c < columns; c++)
    for (r = 0; r < rows; r++)
      sum += table[c * columns + r];
```

Simulates traversal sequential access

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

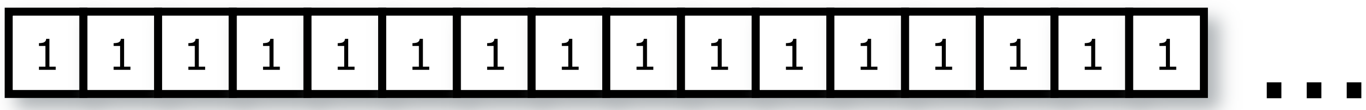
A Simple C++

- Logical



- Physical

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



Demo

In-Memory Databases

Jens Krueger

Enterprise Platform and Integration Concepts Group

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)

Differences to disk-based systems

- Volatile
- Direct access
- Access time
- Access cost

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?

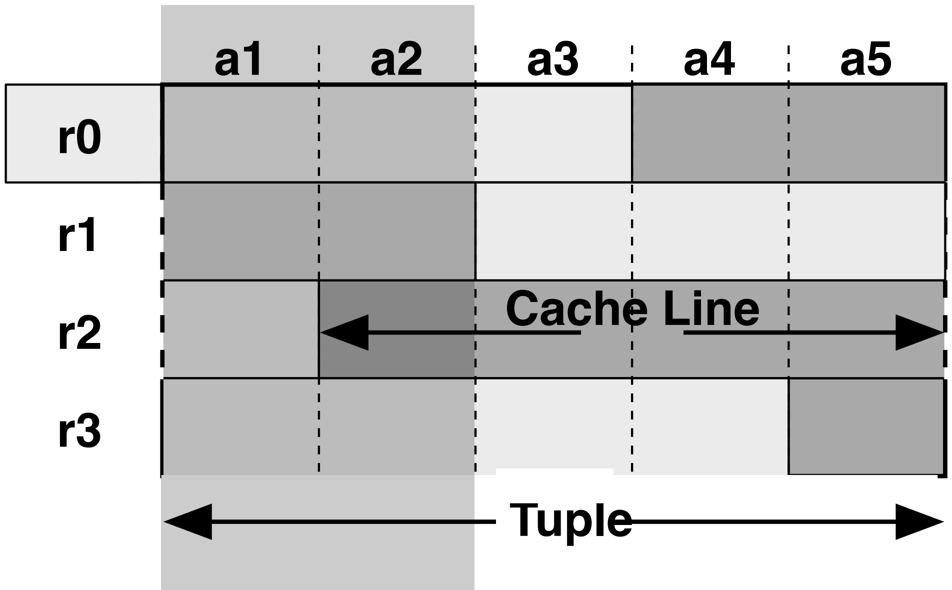
What is the difference between an IMDB and a disk-based DB with a large cache?

- 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.

IMDB: Relations and Cache Lines

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



Question

How to optimize an IMDB?

Question + Answer

How to optimize an IMDB?

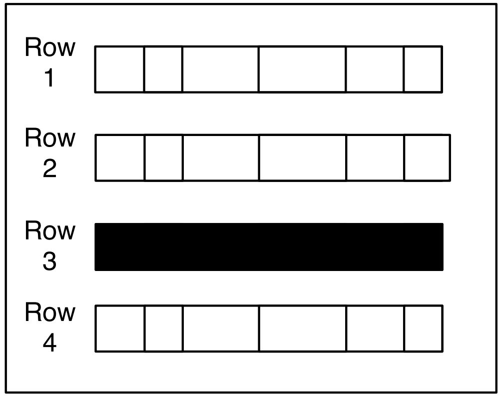
- Exploit sequential access
- Leverage locality

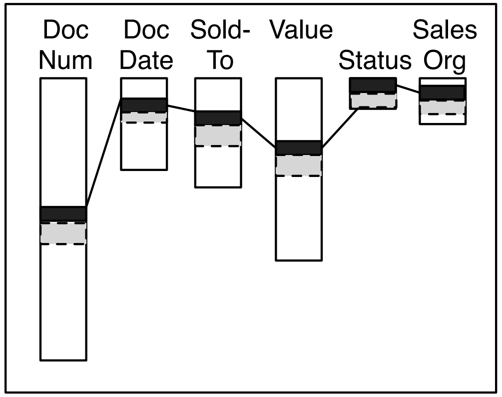
Row- or Column-oriented Storage

Row Store

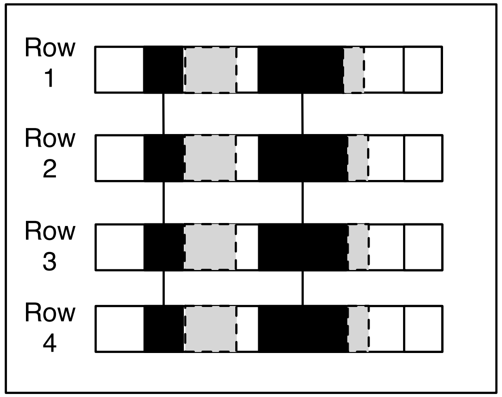
Column Store

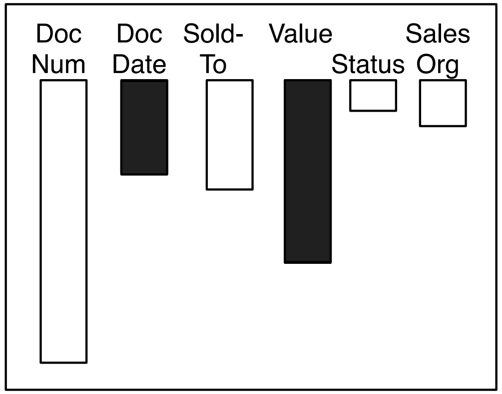
SELECT *
FROM Sales Orders
WHERE Document Number = ‘95779216’





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

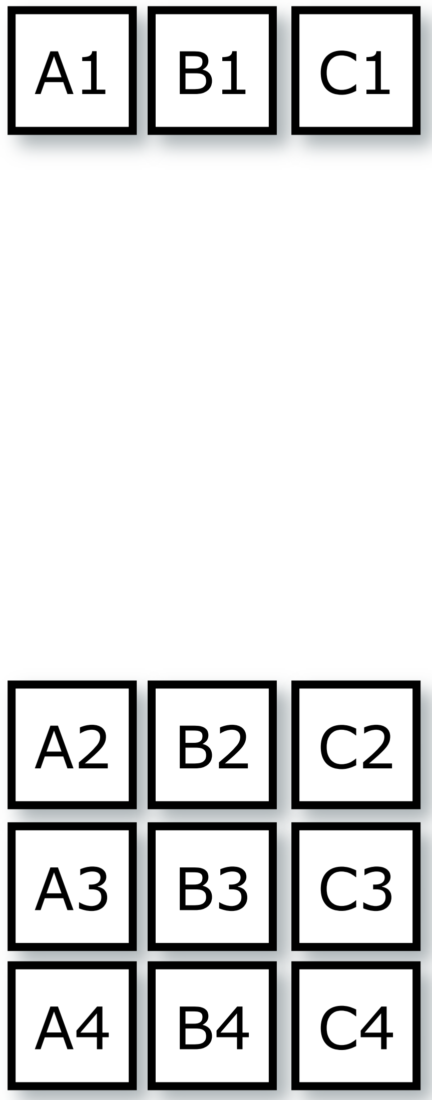




Row-oriented storage

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

Row-oriented storage



Row-oriented storage

A1 B1 C1 A2 B2 C2

A3 B3 C3

A4 B4 C4

Row-oriented storage

A1 B1 C1 A2 B2 C2 A3 B3 C3

A4 B4 C4

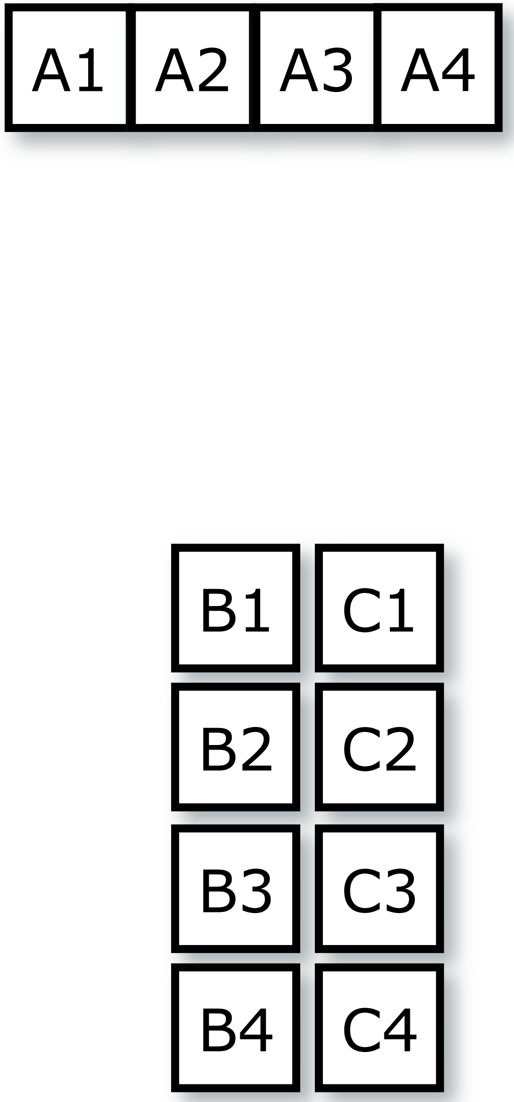
Row-oriented storage

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

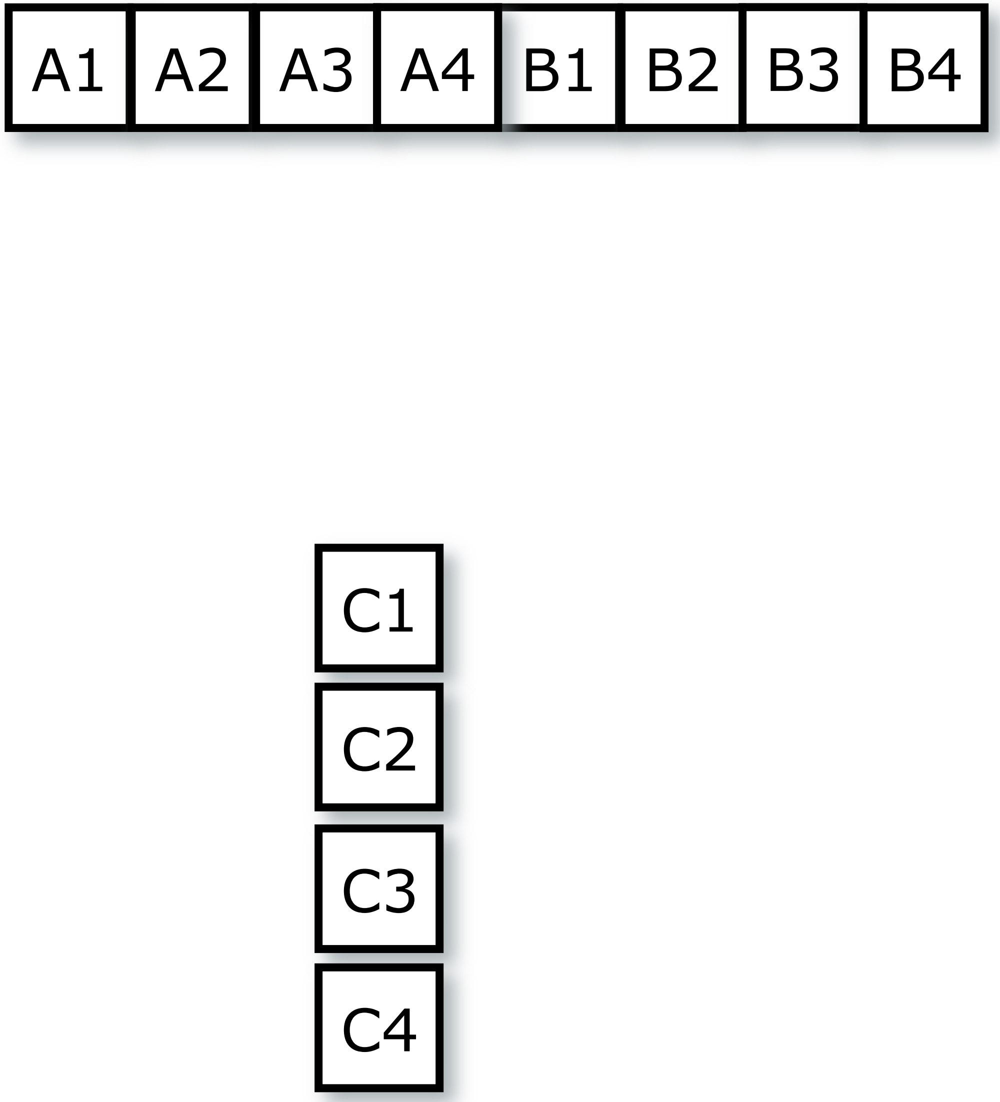
Column-oriented storage

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

Column-oriented storage



Column-oriented storage

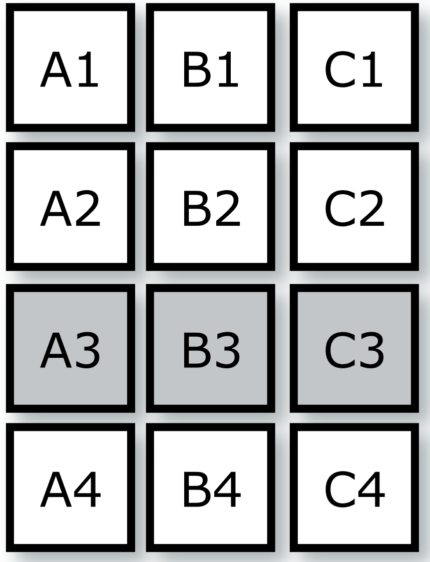


Column-oriented storage

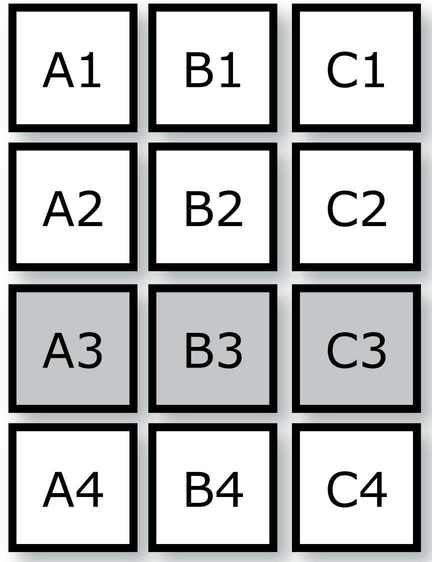
A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	C3	C4

Example: OLTP-Style Query

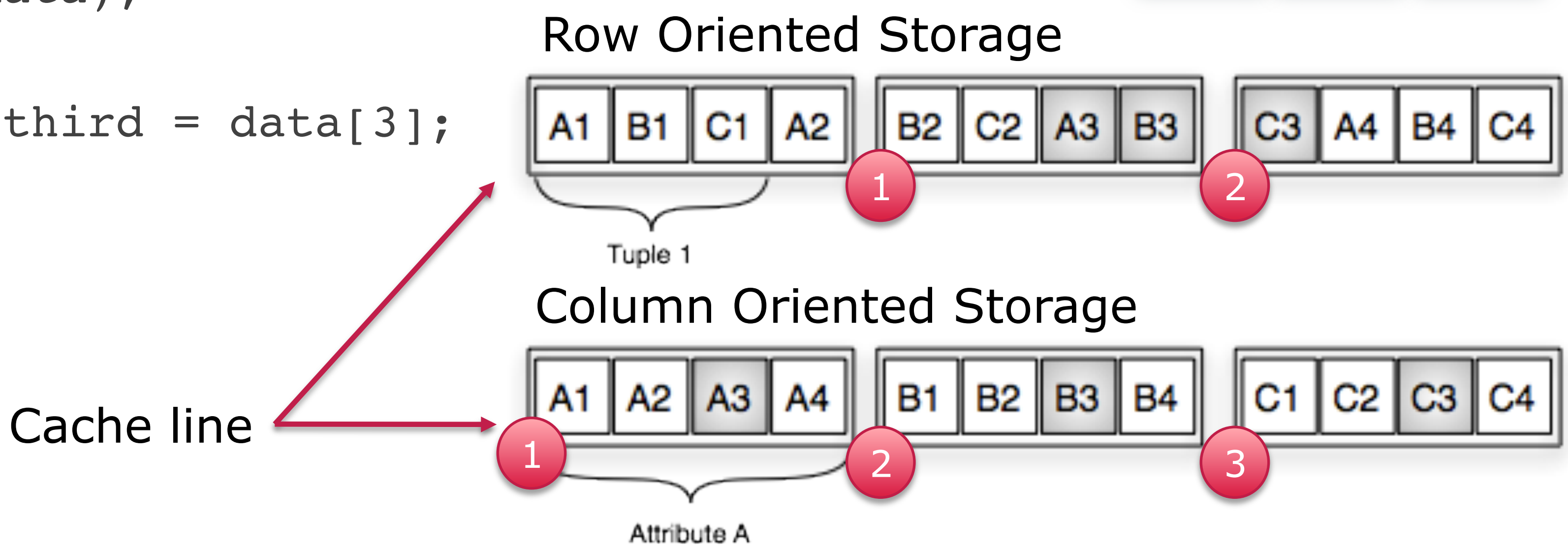
```
struct Tuple {
int a,b,c;
};

Tuple data[4];
fill(data);

Tuple third = data[3];
```



Example: OLTP-Style Query

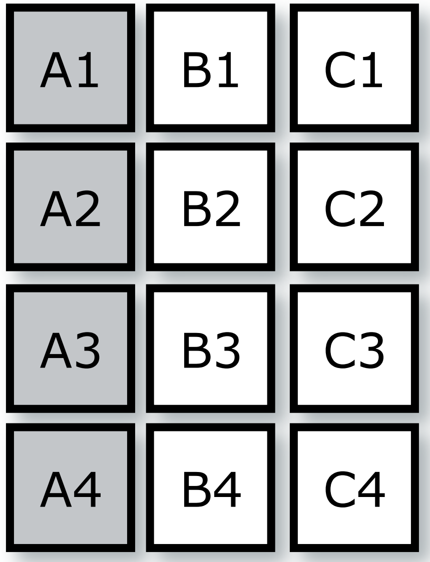



```
struct Tuple {
int a,b,c;
};

Tuple data[4];
fill(data);

Tuple third = data[3];
```



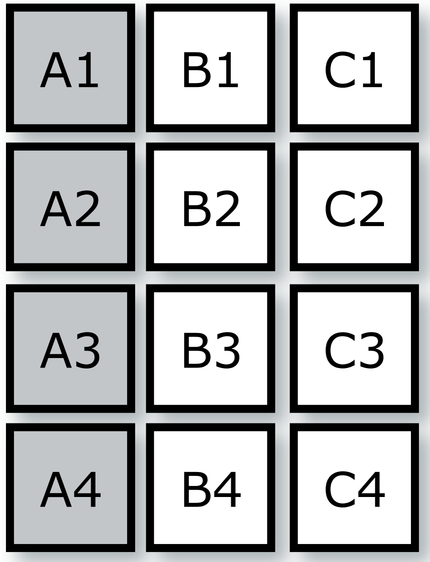
Example: OLAP-Style Query

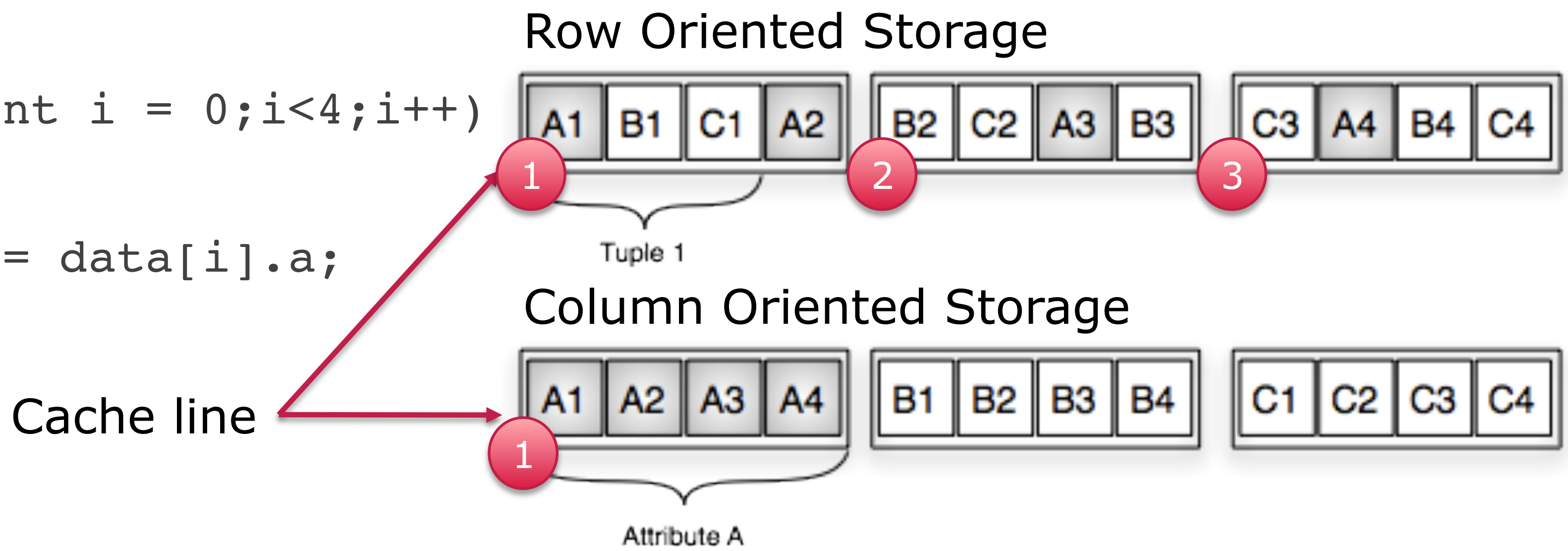
```
struct Tuple {
int a,b,c;
};

Tuple data[4];
fill(data);

int sum = 0;

for(int i = 0;i<4;i++)

sum += data[i].a;
```



Example: OLAP-Style Query

```
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;
```

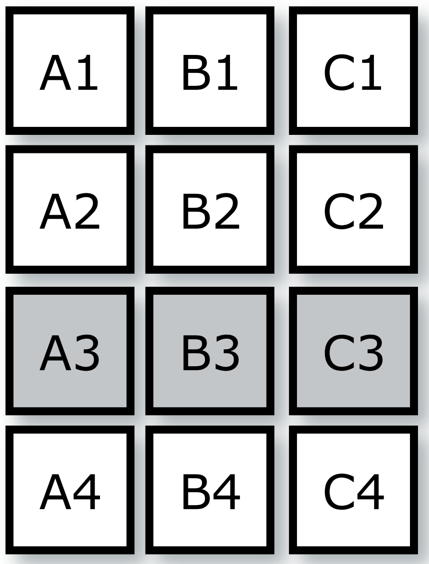




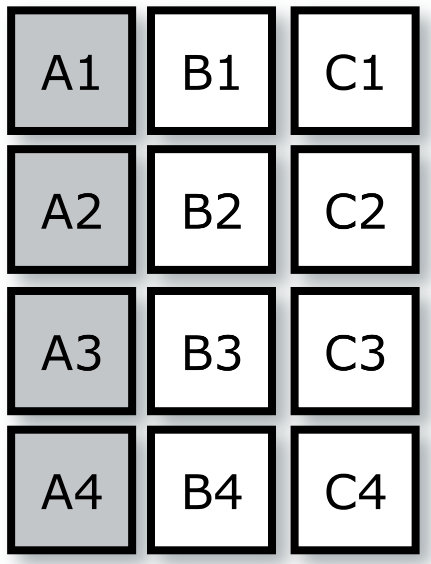
Mixed Workloads

- Mixed Workloads involve attribute- and entity-focused queries

OLTP-style queries



OLAP-style queries

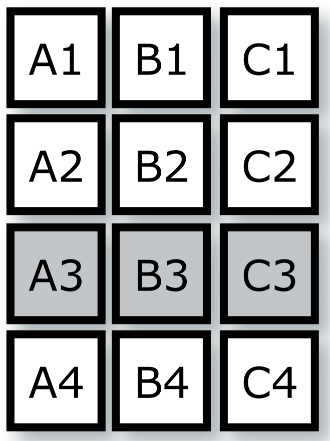


HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

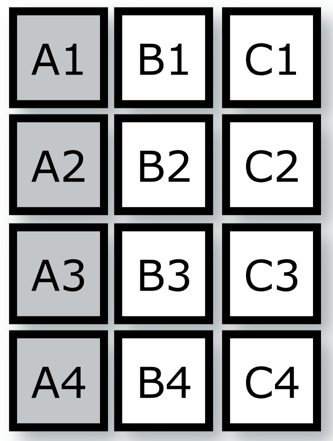
Mixed Workloads: Choosing the Layout

Layout	OLTP-Misses	OLAP-Misses	Mixed
Row	2	3	5
Column	3	1	4

OLTP-style queries



OLAP-style queries



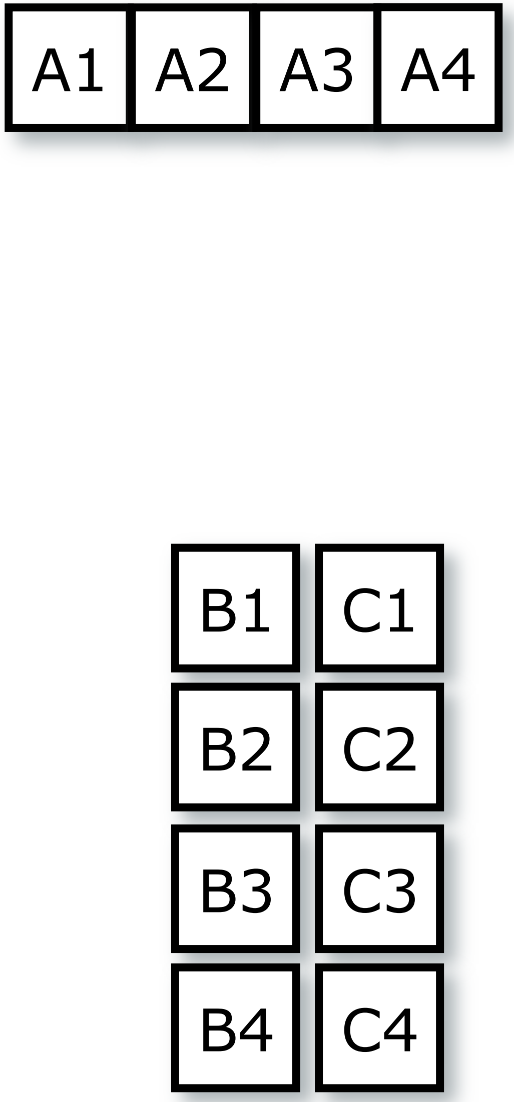
Question

What is the best layout for mixed workloads?

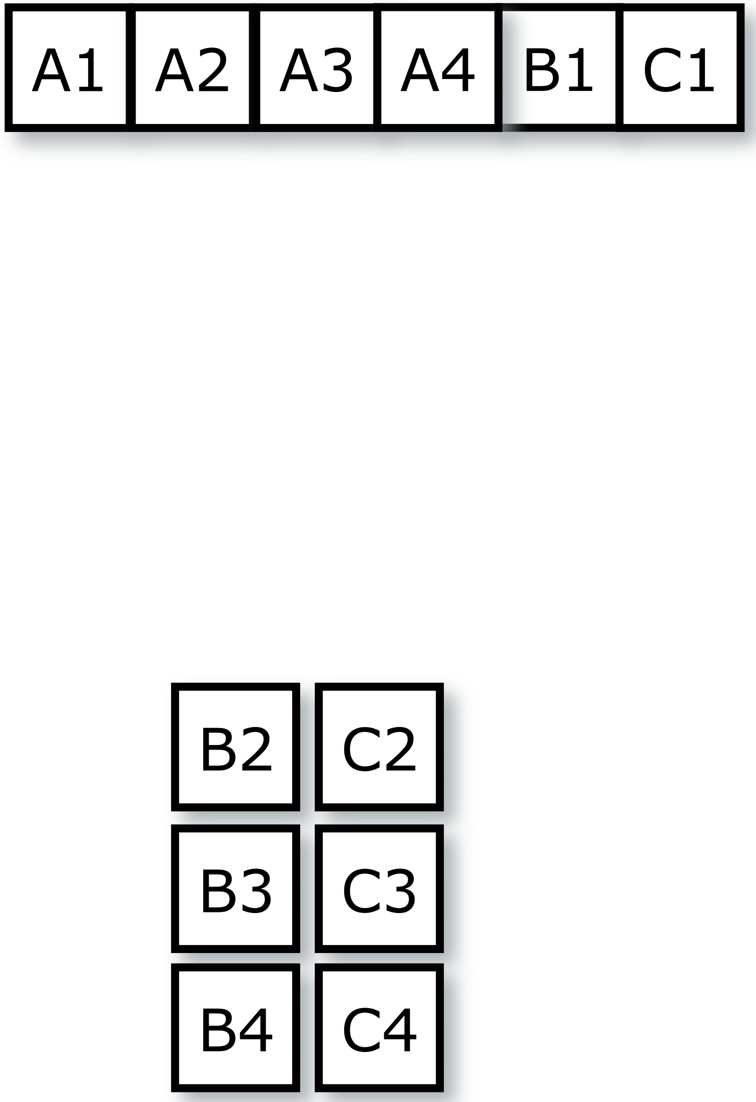
Hybrid-oriented storage

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

Hybrid-oriented storage



Hybrid-oriented storage

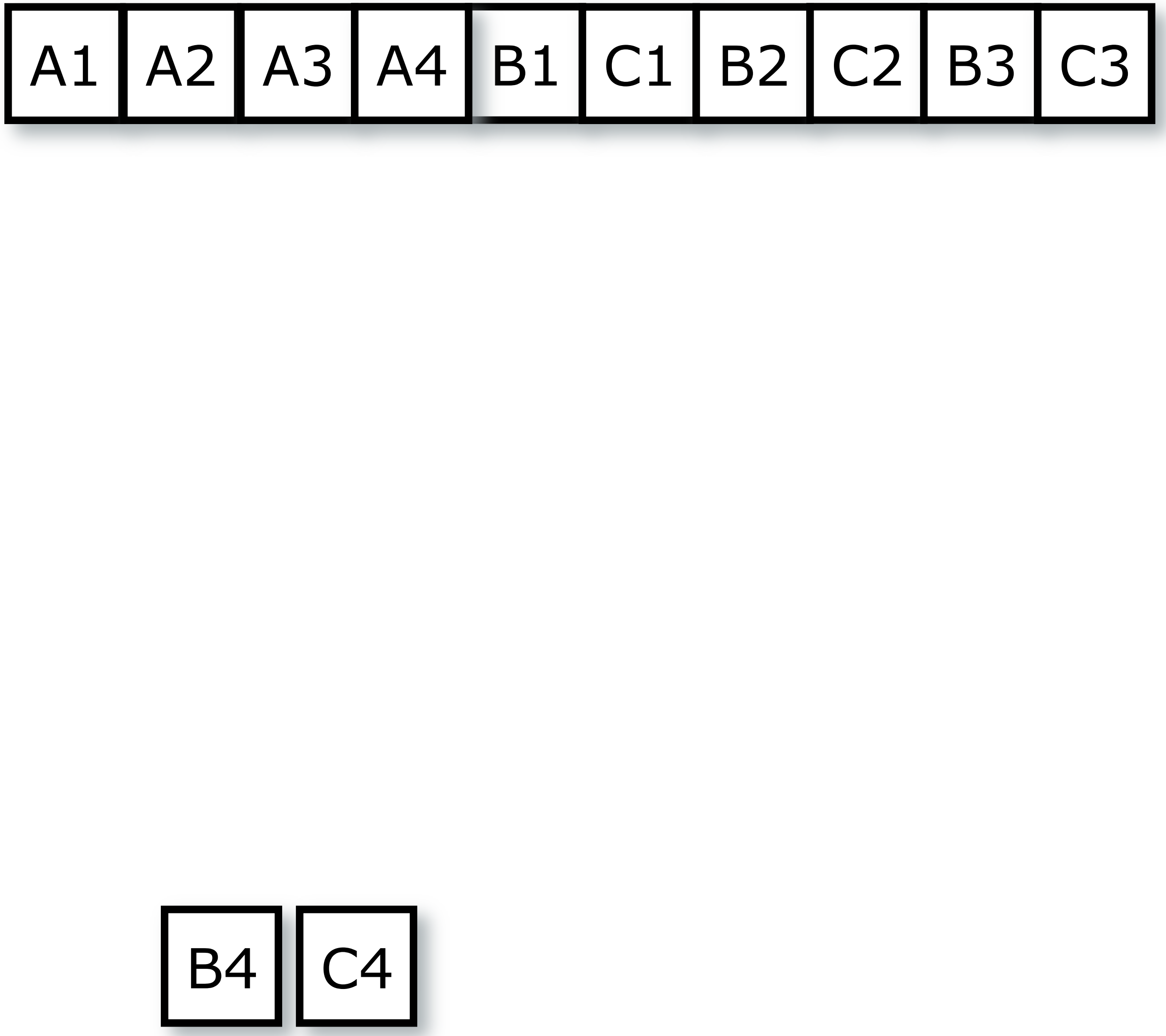


Hybrid-oriented storage

A1	A2	A3	A4	B1	C1	B2	C2

B3	C3
B4	C4

Hybrid-oriented storage

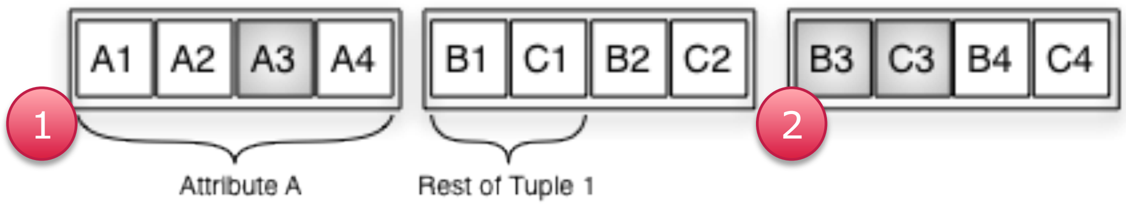


Hybrid-oriented storage

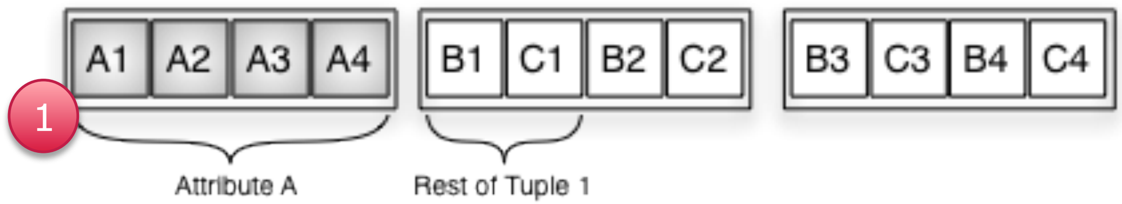
A1	A2	A3	A4	B1	C1	B2	C2	B3	C3	B4	C4

Hybrid: Grouping of Columns

Access tuple 3



Query attribute A



Layout	OLTP-Misses	OLAP-Misses	Mixed
Row	2	3	5
Column	3	1	4
Hybrid	**2**	**1**	**3**

What other optimization for an IMDB?

Compression In Databases

Jens Krueger

Enterprise Platform and Integration Concepts Group

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

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

RecId 1	JAN	INTEL	RED	€1
RecId 2	FEB	ABB	GREEN	€2
RecId 3	MAR	HP	BLUE	€2
RecId 4	APR	INTEL	RED	€4
RecId 5	MAY	IBM	WHITE	€5
RecId 6	JUN	IBM	BLACK	€5
RecId 7	JUL	SIEMENS	BROWN	€4
RecId 8	AUG	INTEL	BLUE	€3
…	…	…	…	…

Dictionary

ValueId	Value
1	INTEL
2	ABB
3	HP
4	IBM
5	SIEMENS

Attribute Table

RecId	ValueId
2	1
3	2
4	3
5	1
10	4
11	4
12	5
13	1

Index

ValueId	RecIdList
2	1,4,8
3	2
4	3
5	5,6

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

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 - log2(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
  - value1 < value2 <=> valueID1 < valueID2

Materialization in Column Stores

Jens Krueger

Enterprise Platform and Integration Concepts Group

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:

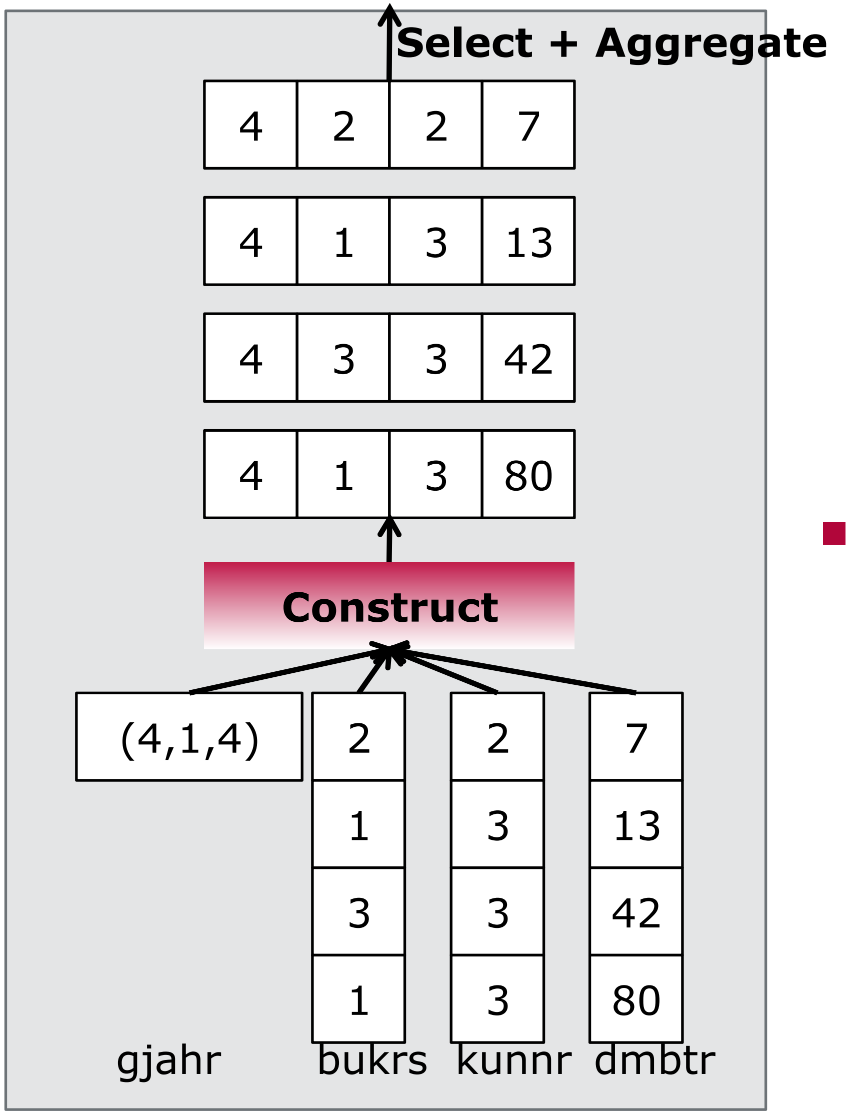
Query:

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

Table BSEG

gjahr	bukrs	kunnr	dmbtr
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

Early materialization



Reference: D. Abadi: SIGMOD 2009

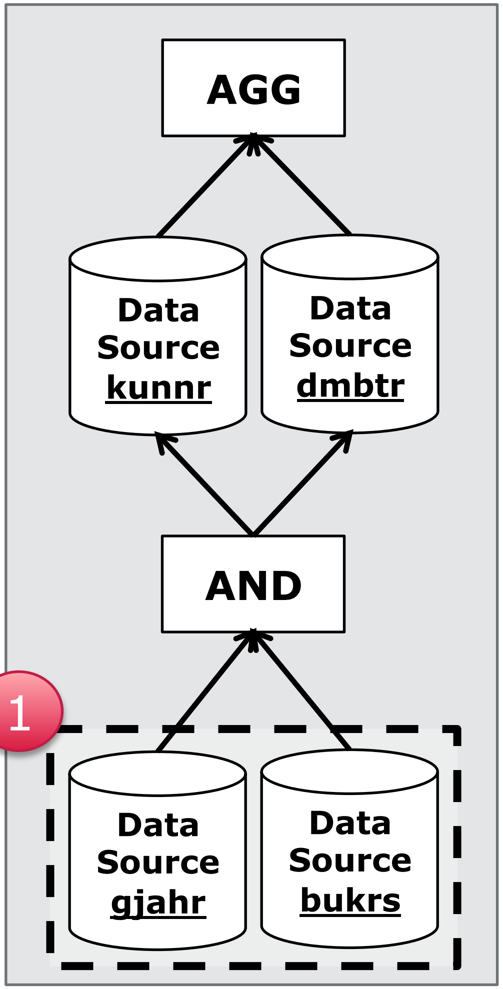
Query:

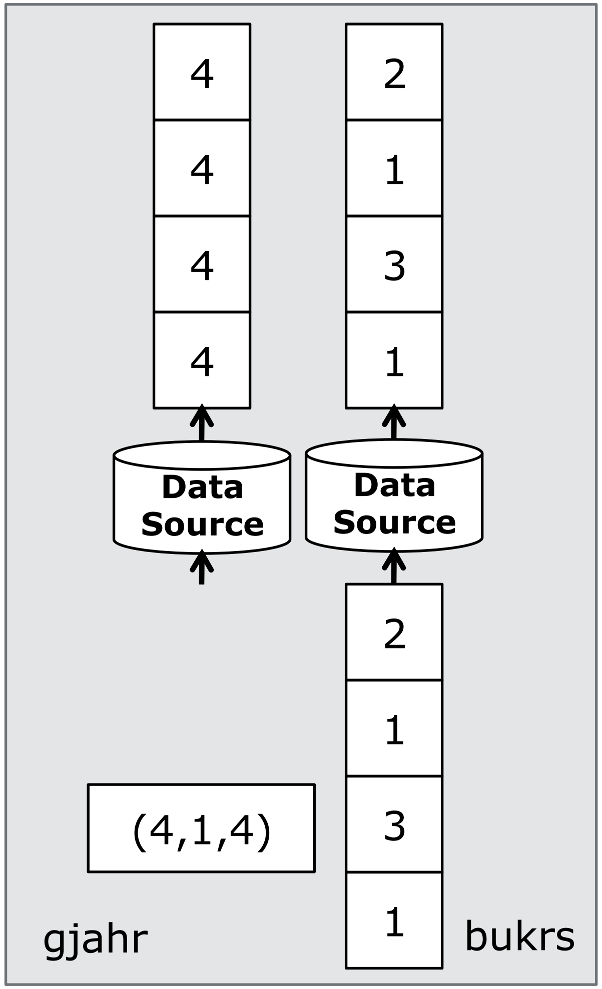
SELECT kunnr, sum(dmbtr)
FROM BSEG
WHERE gjahr = 4
AND bukrs = 1
GROUP BY kunnr

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

Late materialization I

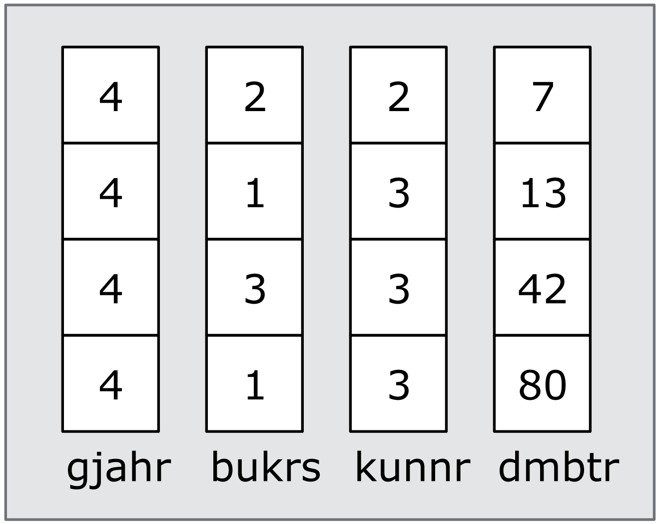
- **Operate on columns**





Query:

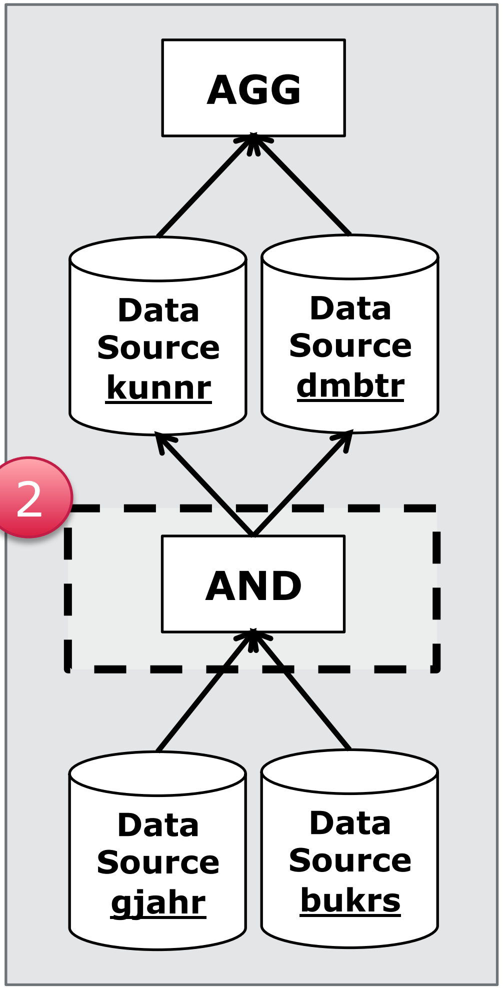
```
SELECT kunnr, sum(dmbtr)
FROM BSEG
WHERE     gjahr = 4
AND       bukrs = 1
GROUP BY kunnr
```

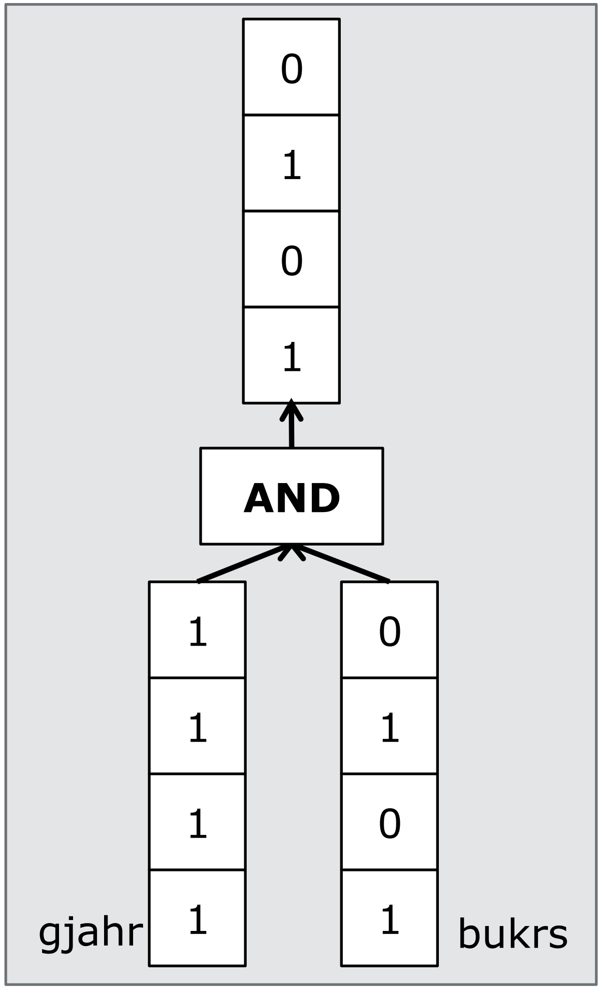


Reference: D. Abadi: SIGMOD 2009

Late materialization II

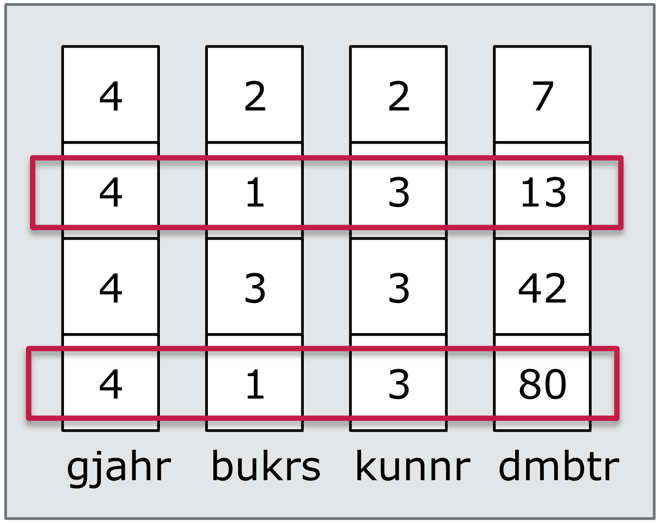
- **Operate on columns**





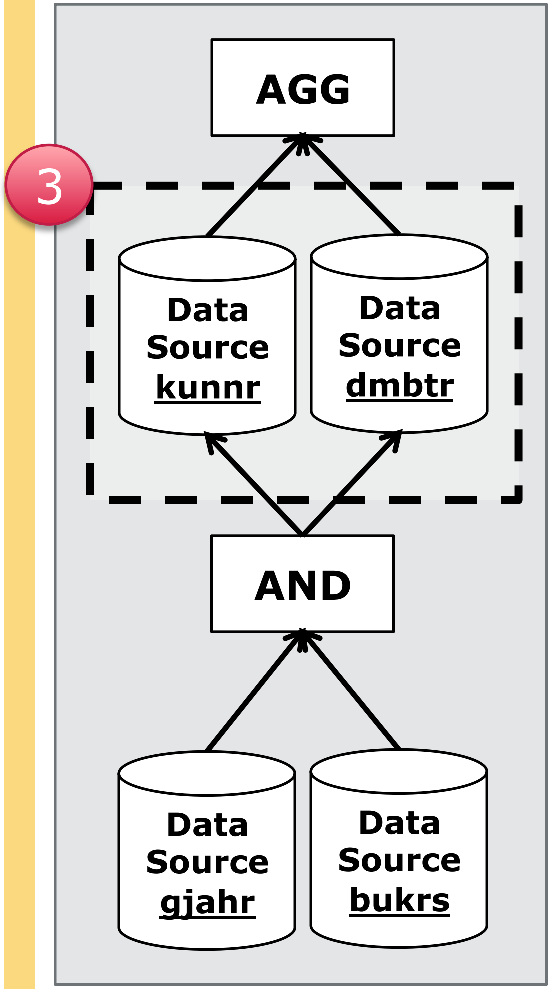
Query:

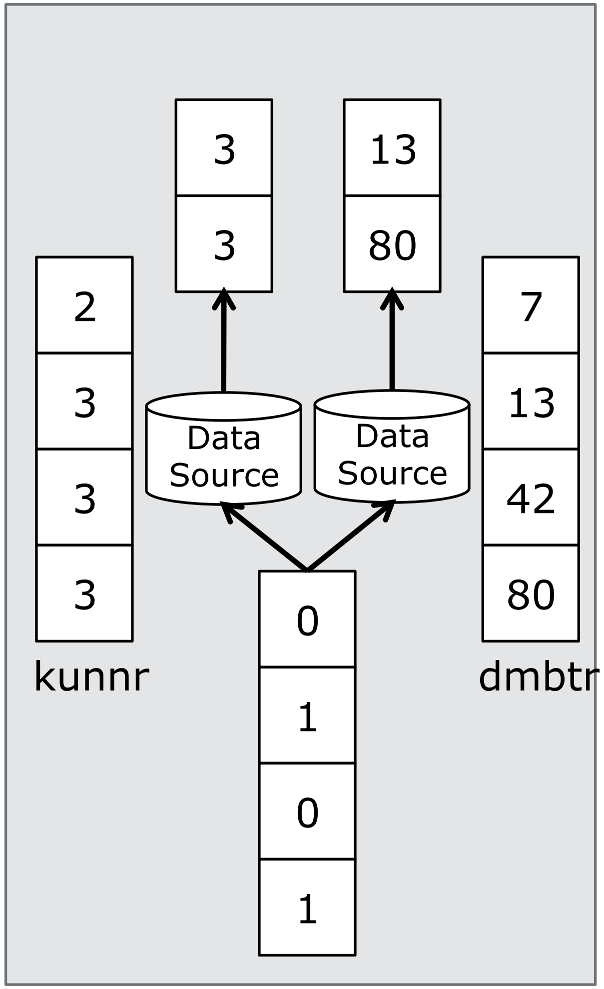
SELECT kunnr, sum(dmbtr)
FROM BSEG
WHERE gjahr = 4
AND bukrs = 1
GROUP BY kunnr



Late materialization III

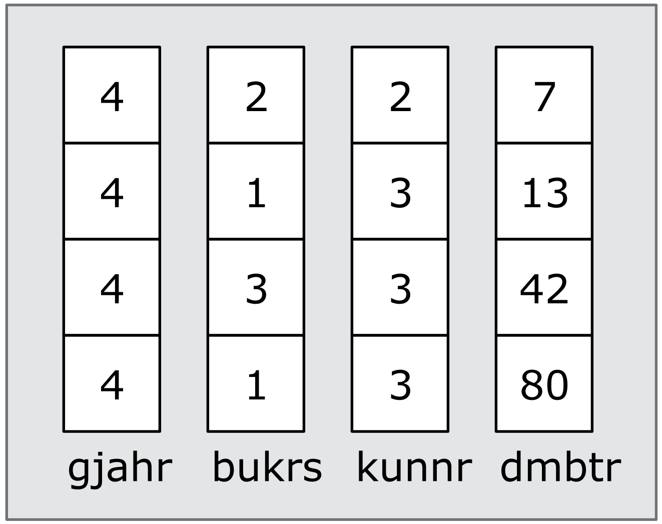
- **Operate on columns**





Query:

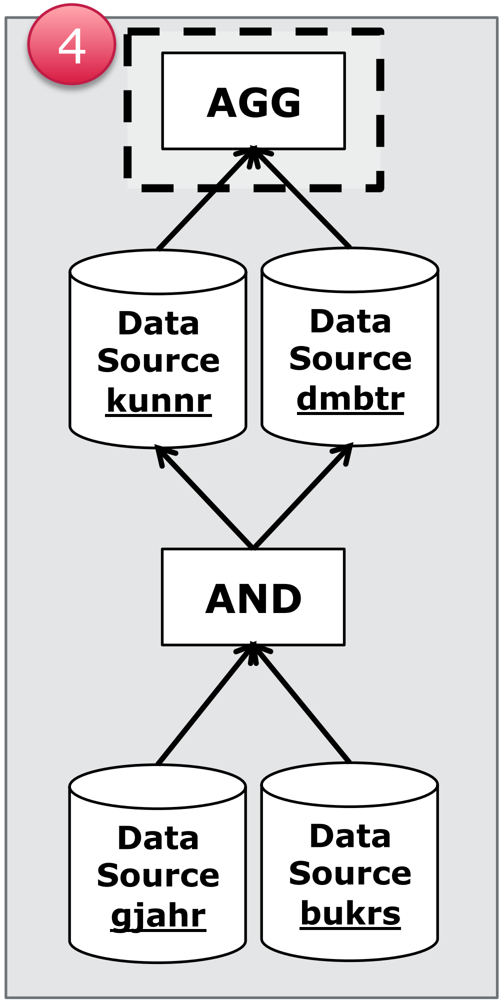
SELECT kunnr, sum(dmbtr)
FROM BSEG
WHERE gjahr = 4
AND bukrs = 1
GROUP BY kunnr

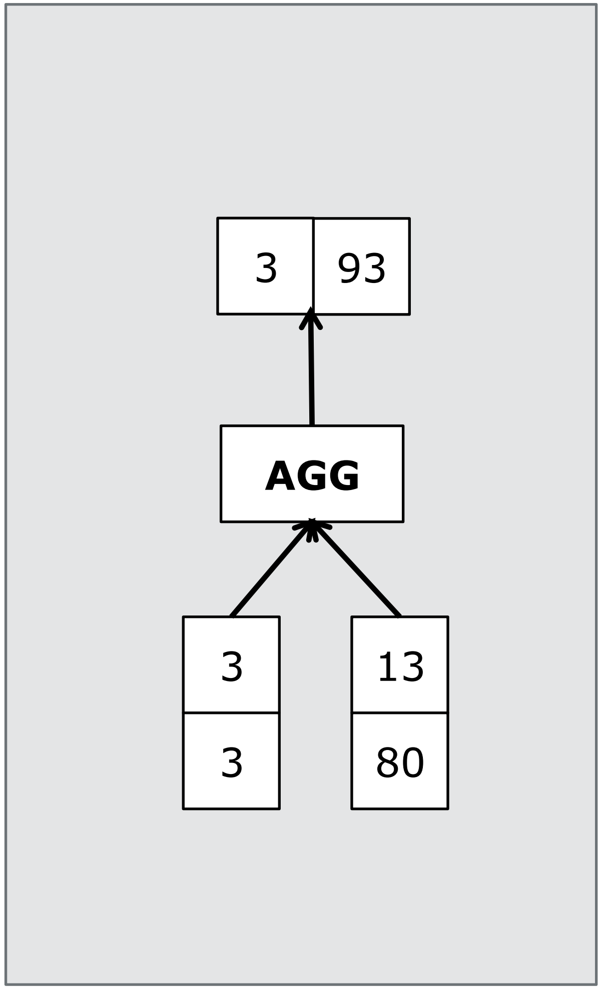


Reference: D. Abadi: SIGMOD 2009

Late materialization IV

- **Operate on columns**





Query:

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

gjahr	bukrs	kunnr	dmbtr
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80

Reference: D. Abadi: SIGMOD 2009