In-Memory Databases

Jens Krueger

Enterprise Platform and Integration Concepts
Hasso Plattner Intitute

Outline

- Recap on Memory Access
- Motivation for a new DBMS architecture
- Overview on Physical Data Organization
- In-Memory Databases
- Column-Store Optimizations

Recap: Memory Access

Jens Krueger

Enterprise Platform and Integration Concepts
Hasso Plattner Intitute

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
Magnetic Disk		**~ 10 000 000 ns** (10 ms)	**TB**

Data Processing

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

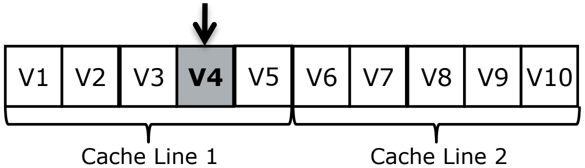
- Not CPU bound
- **But** bandwidth bound
- "I/O Bottleneck"

→ 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 Hierarchy

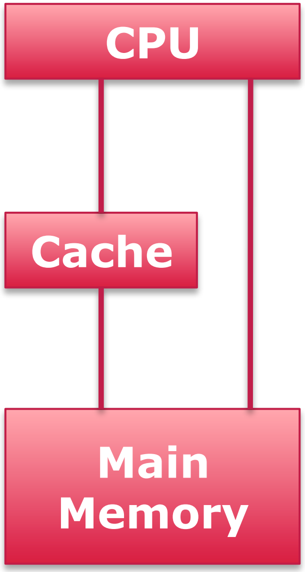
- **Cache**

  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**).

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)

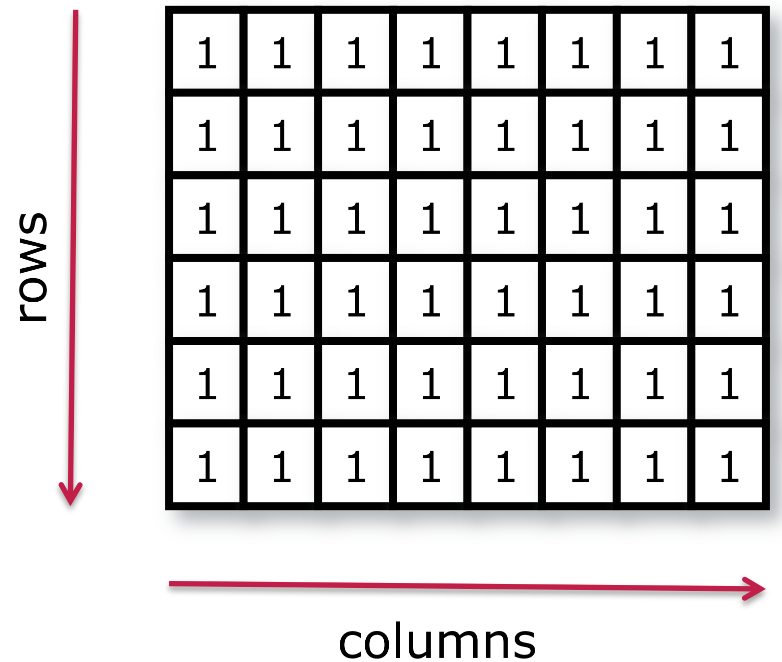
To improve locality

- 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/information into a cache line
- Random access (hash join):
  - Partition to fit in cache (cache-sized hash tables)

HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

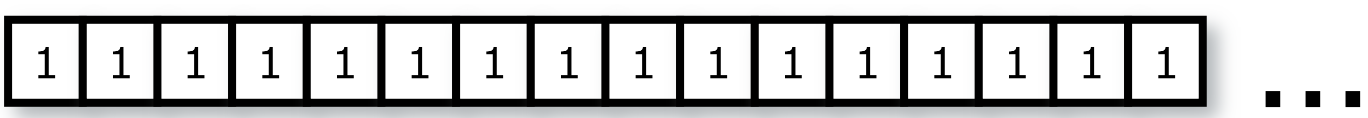
A Simple C++

- Logical

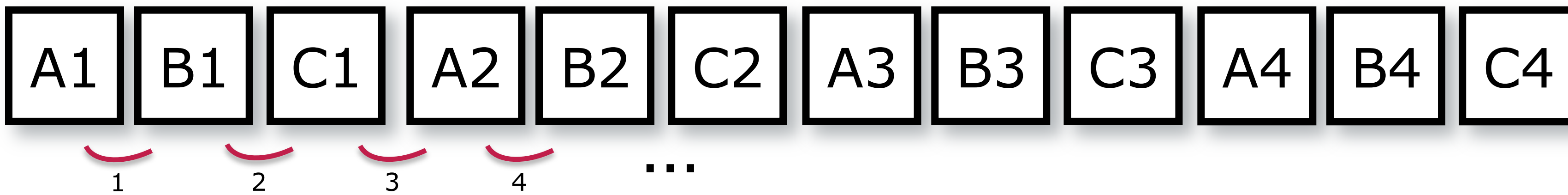


- Physical

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



Example for Sequential Access

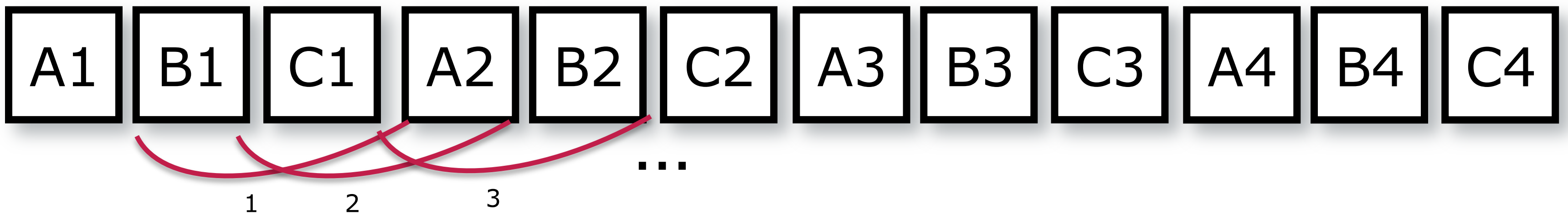



```
for (r = 0; r < rows; r++)
    for (c = 0; c < columns; c++)
        sum += table[r * columns + c];
```

Simulates sequential access

- All data in a cache line is read
- Prefetching and pipelining further **improve** performance

Example for Traversal Sequential Access



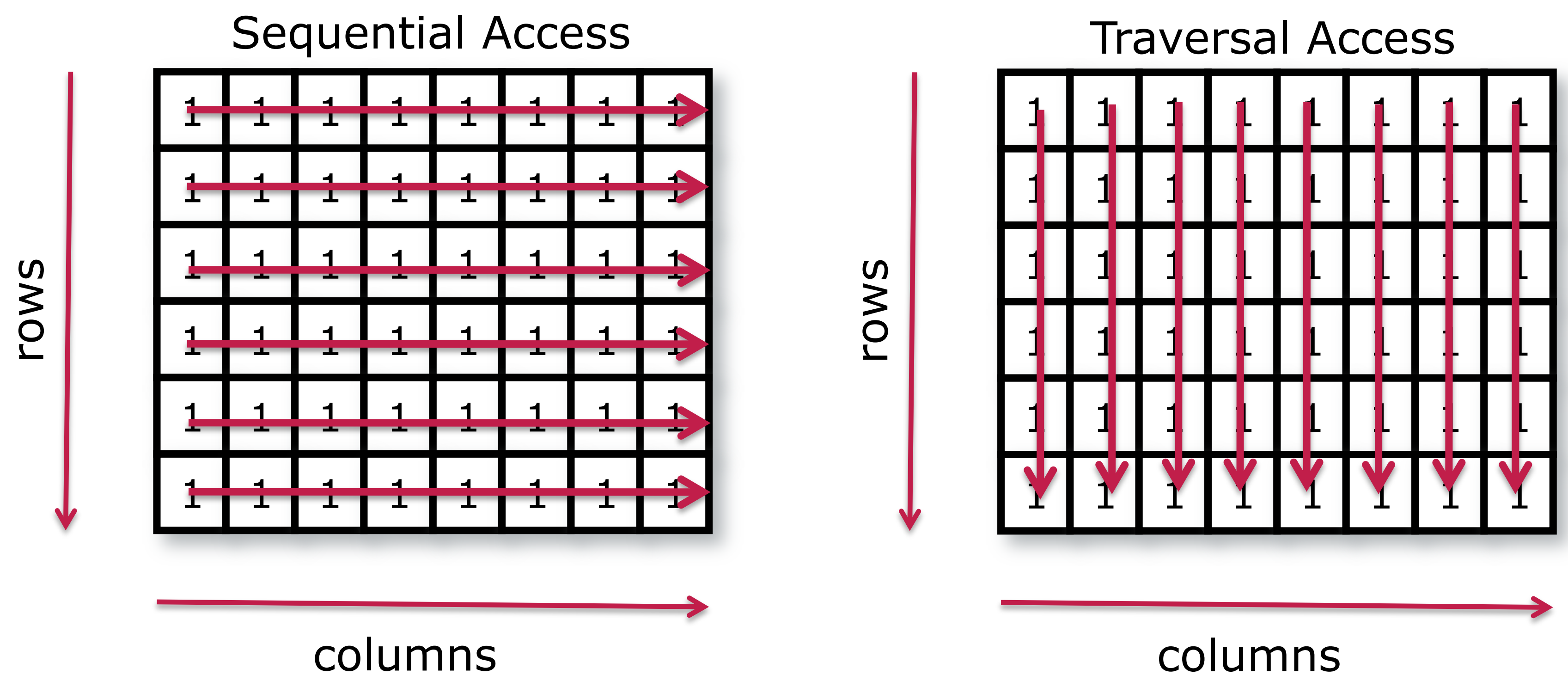

```
for (c = 0; c < columns; c++)
   for (r = 0; r < rows; r++)
     sum += table[c * columns + r];
```

Simulates traversal sequential access

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

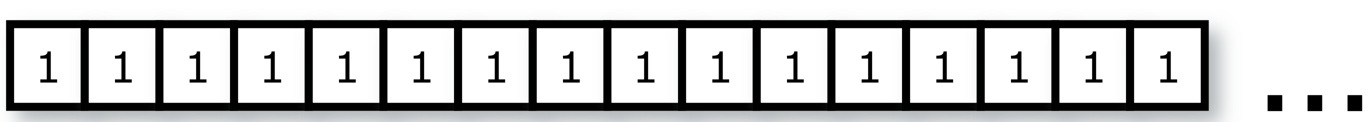
A Simple C++

- Logical



- Physical

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



Do this at home!

Demo C++
for Copy and Paste:

```
//======================================================================
// Name        : micro02.cpp
// Author      : jens
// Description : Aggregation
//======================================================================

#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));

         // fuellen mir Random Int's
         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;


         cout << "\nPress Key: ";
         cin >> w;
/////////////////////////////////////////////////////
         cout << "Sequential Access " << endl;

         seq_sum = 0;
         time = 0;

         // lesen
         gettimeofday(&start1, NULL);

         for (r = 0; r < rows; r++)
            for (c = 0; c < columns; c++)

               //read[c] = table[r * columns + c];
           seq_sum += 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;


         /////////////////////////////////////////////////////
         cout << "Stride Access " << endl;

         stride_sum = 0; time = 0;

         // lesen
         gettimeofday(&start2, NULL);

         for (c = 0; c < columns; c++)
         for (r = 0; r < rows; r++)

               //read[c] = table[r * columns + c];
           stride_sum += table[r * columns + c];

         gettimeofday(&end2, NULL);

         time = (end2.tv_sec - start2.tv_sec) * 1000000 + (end2.tv_usec
               - start2.tv_usec);

         cout << "Sum:  " << stride_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;
}
```

Enterprise-specific Data Management

Jens Krueger

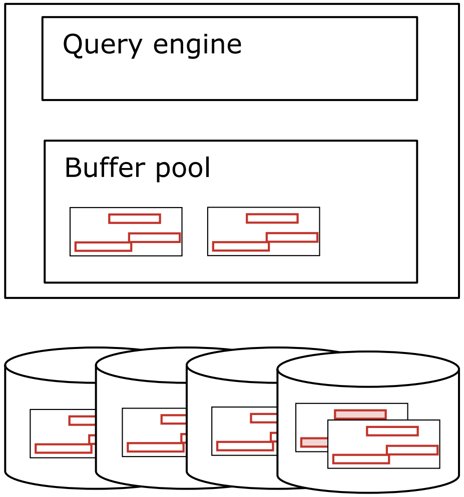
Enterprise Platform and Integration Concepts
Hasso Plattner Intitute

Motivation

- Hardware has changed
  - TB of main memory are available
  - Cache sizes increased
  - Multi-core CPU’s are present
  - Memory bottleneck increased
- Data/Workload
  - Tables are wide and sparse
  - Lots of set processing
- Traditional databases
  - Optimized for write-intensive workloads
  - show bad L2 cache behavior

Problem Statement

- DBMS architecture has **not changed** over decades
- Redesign needed to handle the changes in:
  - Hardware trends (CPU/cache/memory)
  - Changed workload requirements
  - Data characteristics
  - Data amount



Traditional DBMS Architecture

Overview on Physical Data Organization

Jens Krueger

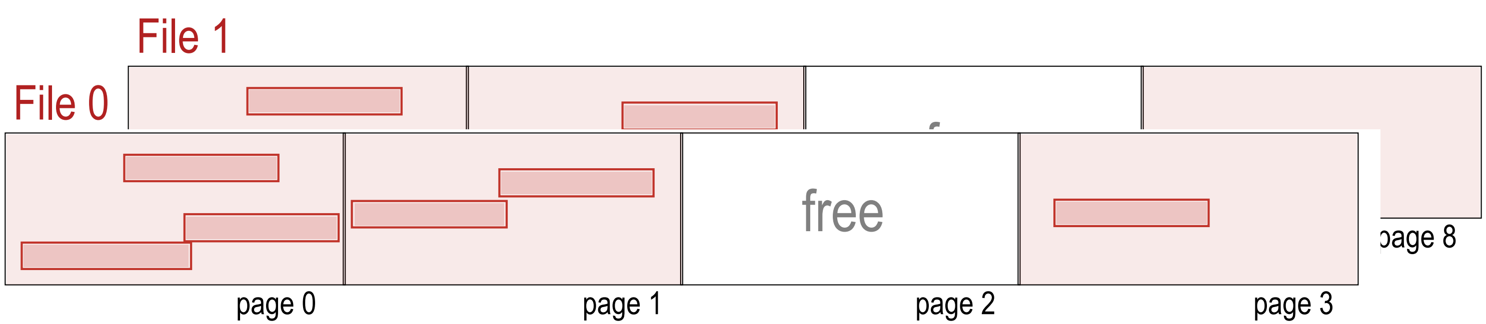
Enterprise Platform and Integration Concepts
Hasso Plattner Intitute

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**)

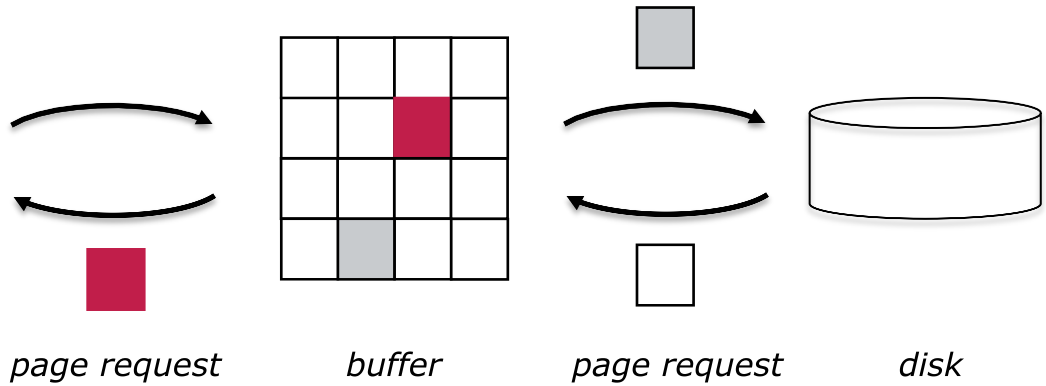
Excursus: Files on Disk

- 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.



Excursus: 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)

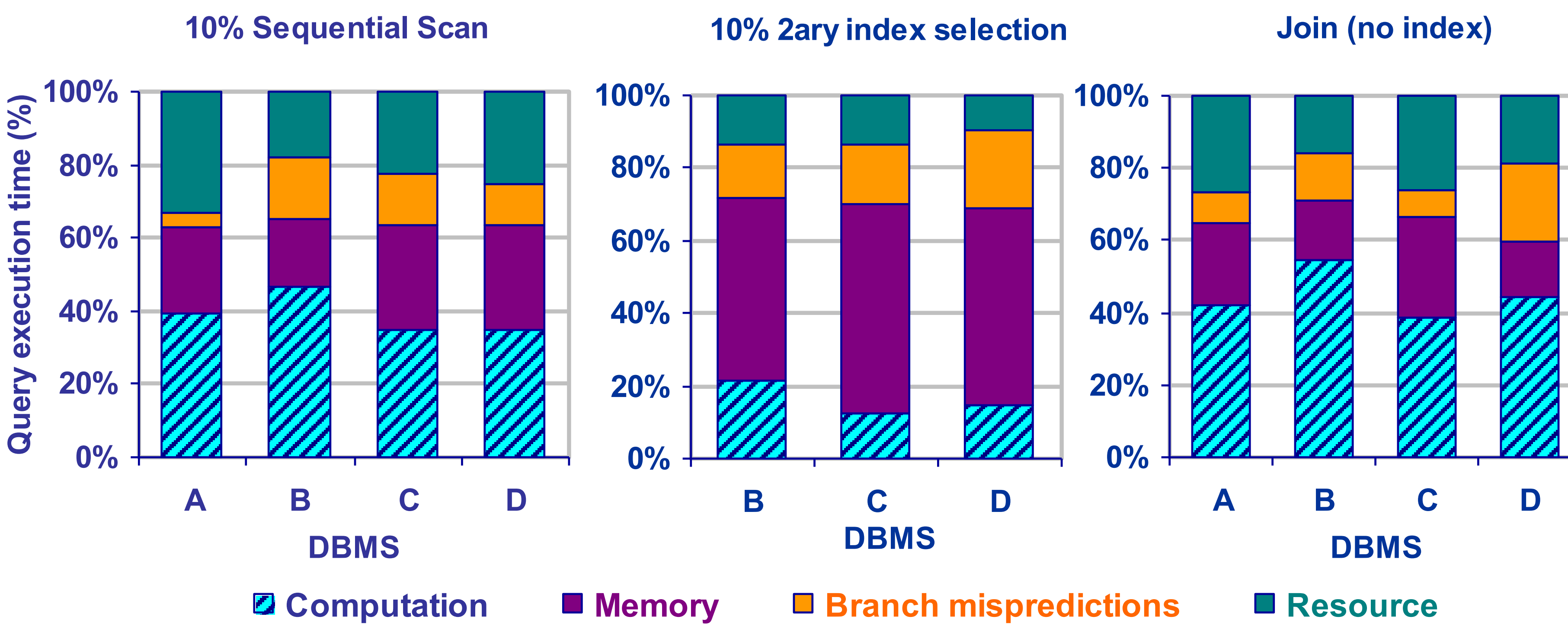


Traditional DBMS: In a Nutshell

- 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 –
  - Bad L2 cache behavior

Problem Statement (contd.)

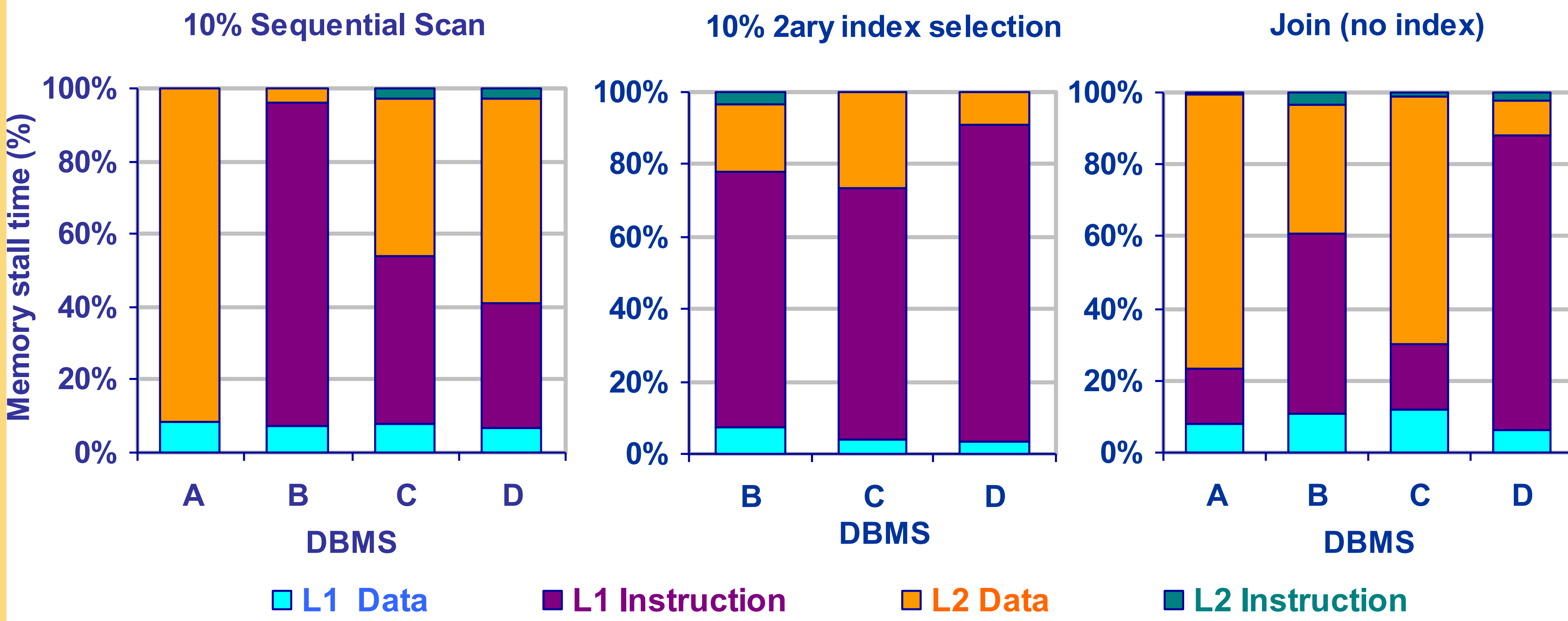
- Traditional DBMS suffer from stalls (>50%)
- Breakdown of execution time:
  (taken from [1])



[1] Ailamaki, A. et.al.: "DBMSs On A Modern Processor: Where Does Time Go?", VLDB 1999

Problem Statement (contd.)

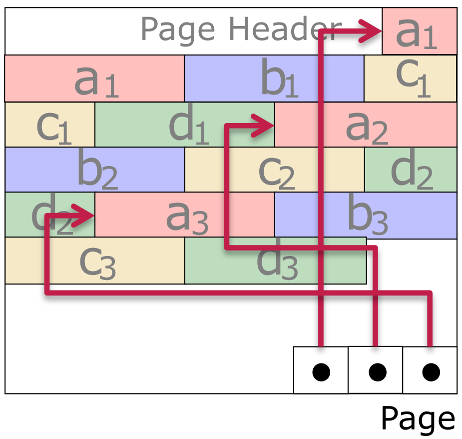
- Breakdown of memory stalls:
  (taken from [1])



[1] Ailamaki, A. et.al.: "DBMSs On A Modern Processor: Where Does Time Go?", VLDB 1999

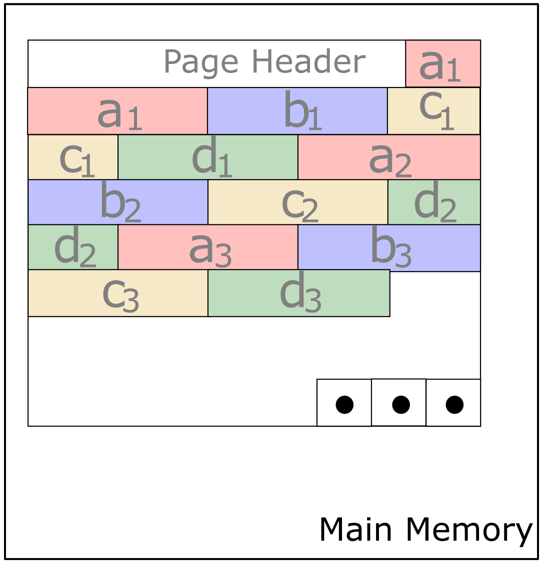
Cache Behavior of a Row Store Page

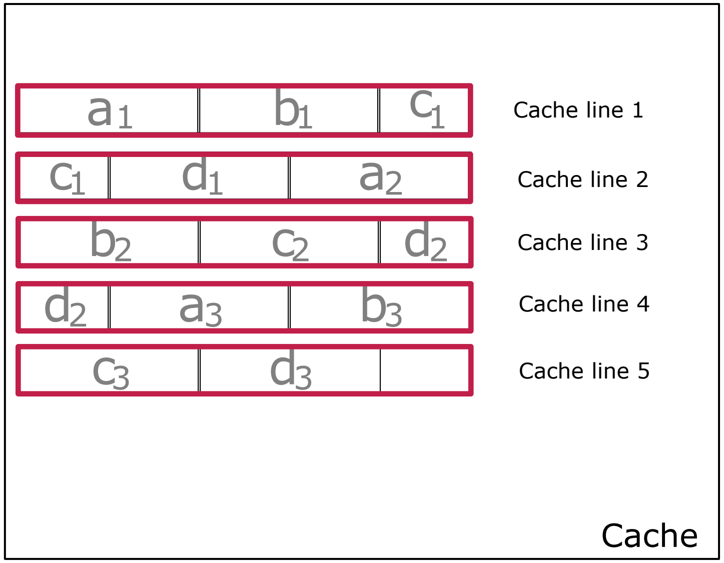
- Pages are mapped to cache lines



Cache Behavior of a Row Store Page

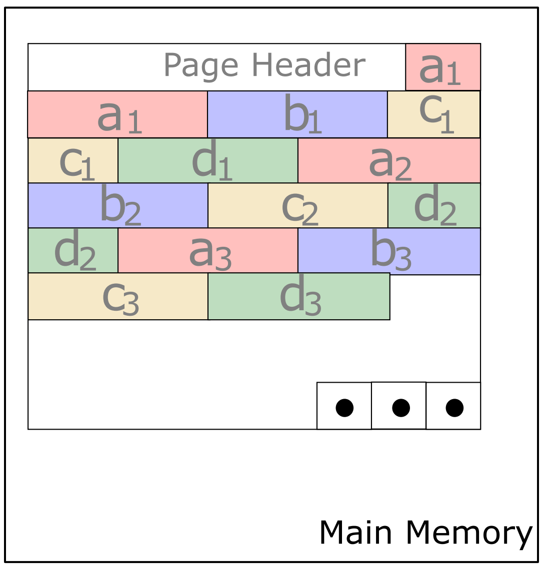
- Pages are mapped to cache lines



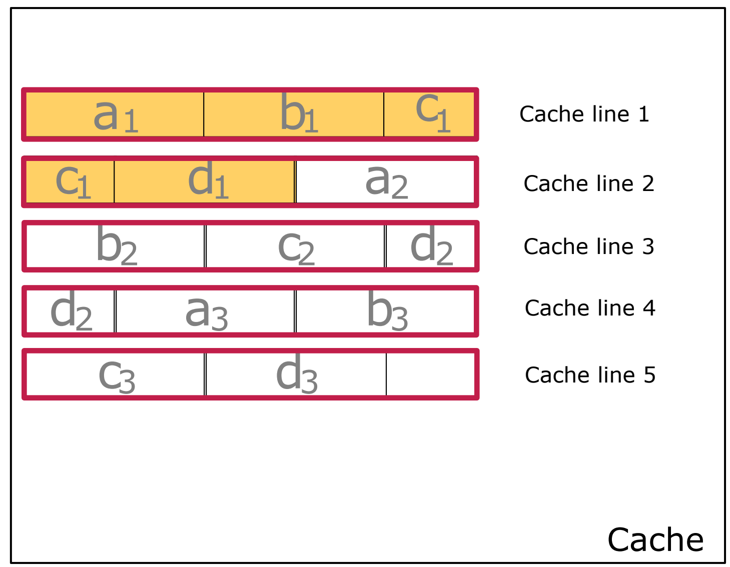


Cache Behavior of a Row Store Page

- Pages are mapped to cache lines

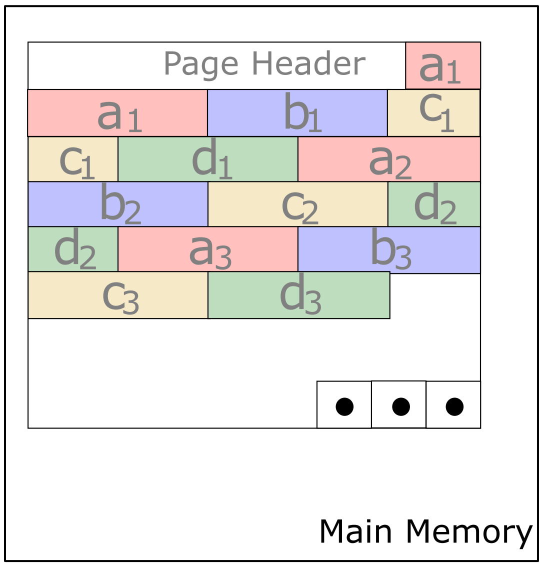


SELECT * FROM table
WHERE a EQ $

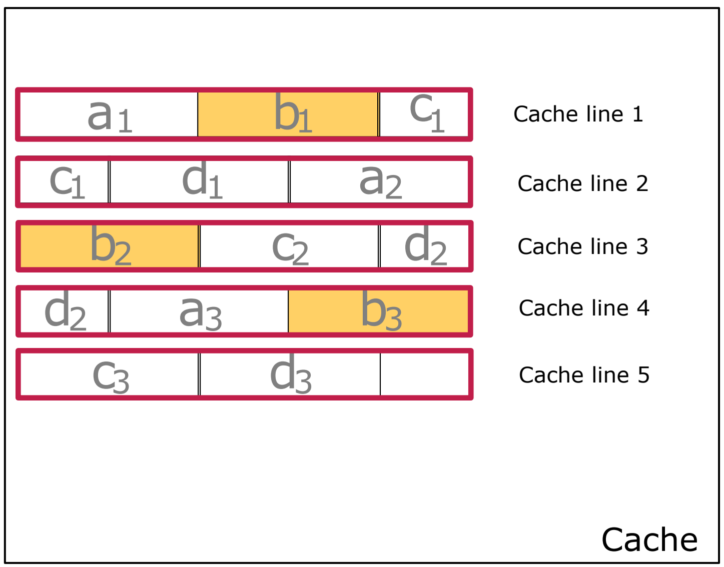


Cache Behavior of a Row Store Page

- Pages are mapped to cache lines

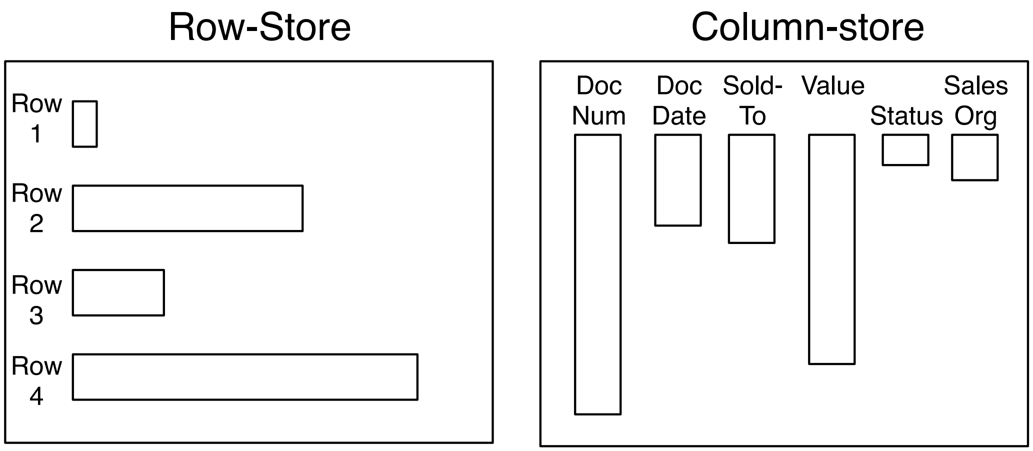


SELECT b FROM table
WHERE b > $



Row-wise vs. Column-wise Data Organization

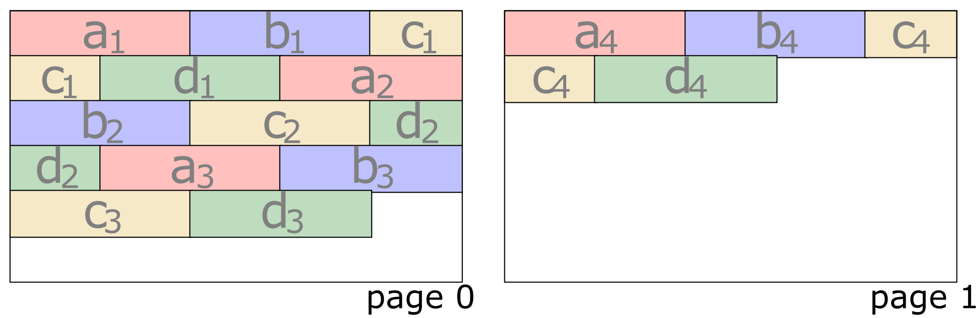
- Row Store:
  - Rows are stored consecutively
  - Optimal for row-wise access (e.g. *)
- Column Store:
  - Columns are stored consecutively
  - Optimal for attribute focused access (e.g.SUM, GROUP BY)
- Note: concept is **independent** form storage variant



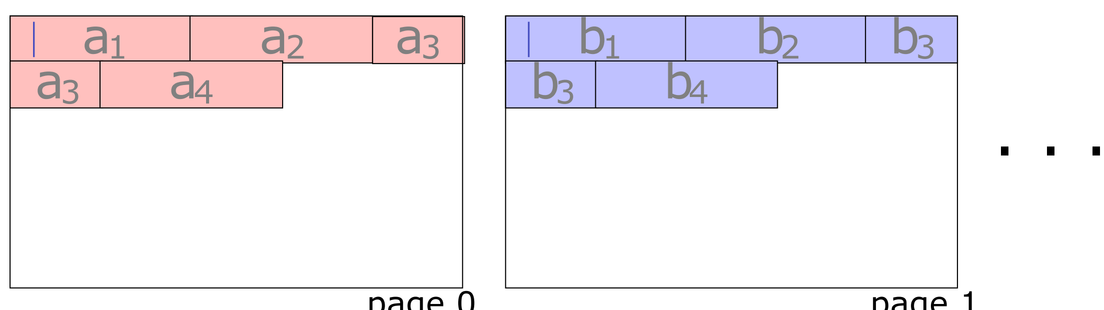
Rows, Columns, and the Page Layout

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)



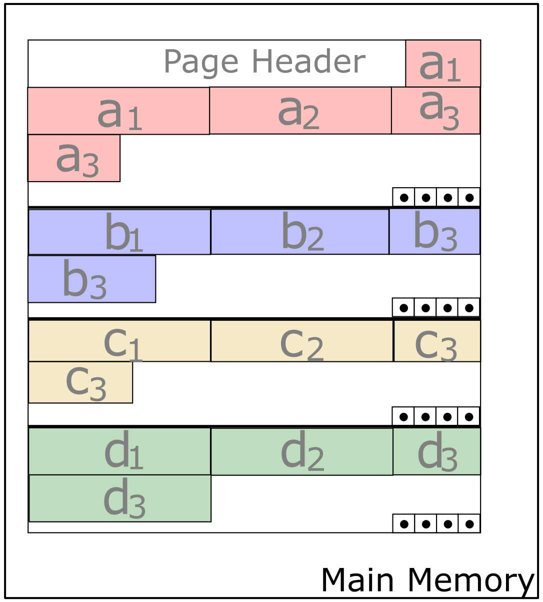
- **Column-oriented page layout** (decomposed storage model)

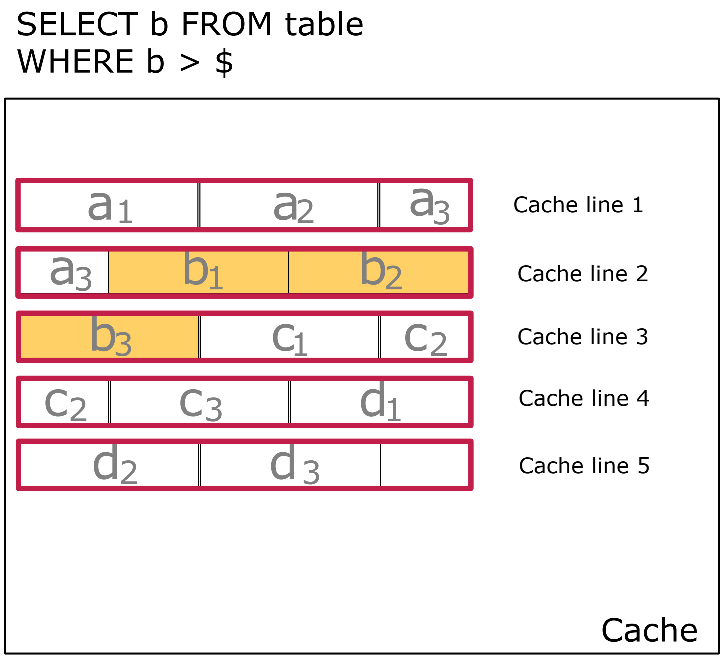


Tuple reconstruction/ decomposition is extremely slow

Partition Attributes Across (PAX)

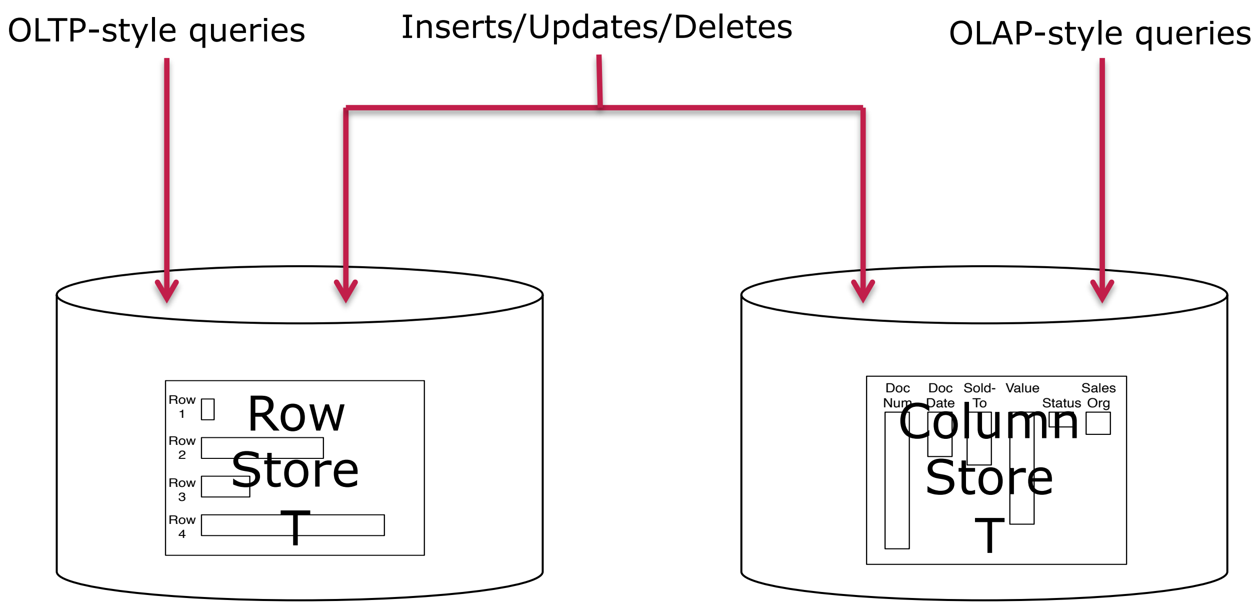
- Concept of column-wise data organization applied **inside** a **single** page
- "Trade-off" for better cache utilization ↑
- But, all columns have to read anyways ↓





Fractured Mirrors or Hybrid Row-Column

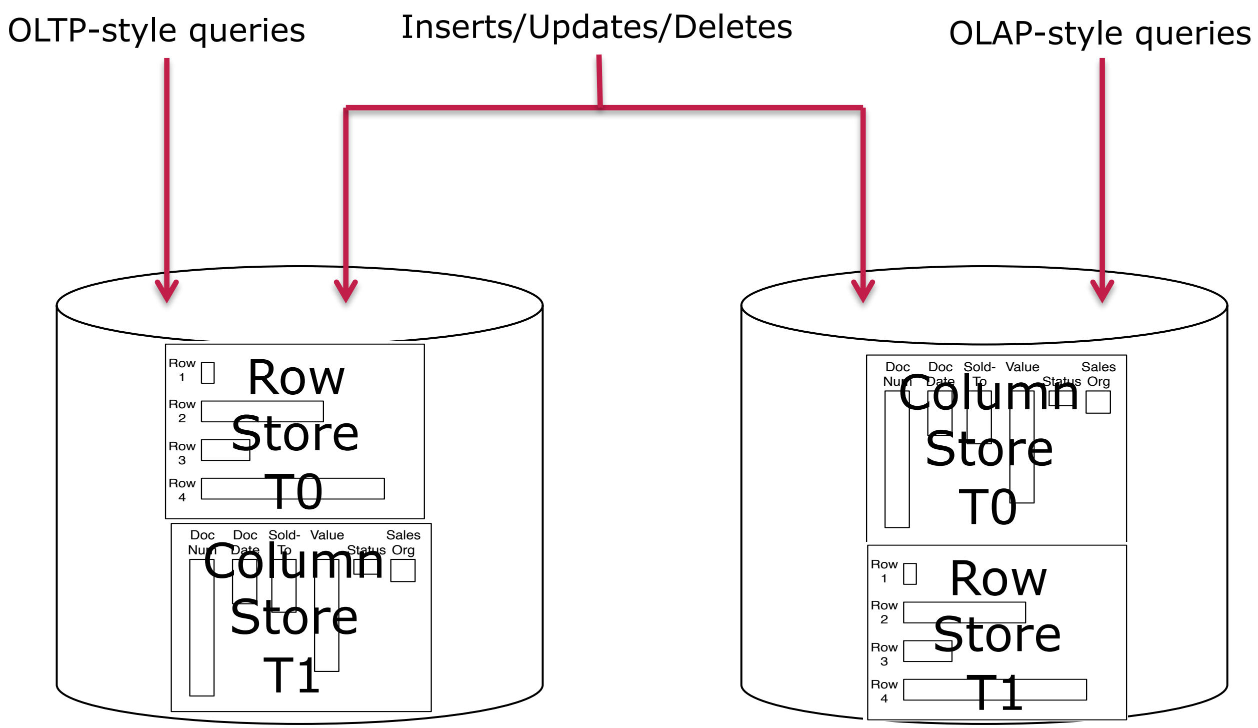
- Both storage concepts are leveraged (see [1], [2])
- Data modifications are applied on both storages ↓
- Data is stored redundantly ↓



[1] Ramamurthy, R. et.al. : *"A case for fractured mirrors"*, VLDB 2003.
[2] Schaffner, J. et.al.: *"A Hybrid Row-Column OLTP Database Architecture for Operational Reporting"*, BIRTE 2008.

Balanced Fractured Mirrors

- Both storage concepts are leveraged (see [1], [2])
- Data modifications are applied on both storages ↓
- Data is stored redundantly ↓
- Round robin for load balancing



[1] Ramamurthy, R. et.al. : *"A case for fractured mirrors"*, VLDB 2003.
[2] Schaffner, J. et.al.: *"A Hybrid Row-Column OLTP Database Architecture for Operational Reporting"*, BIRTE 2008.

Summary

- Disk access
  - Low throughput
  - Slow random access
  - Disk-based Column-stores pollute cache lines with *surrogate ID* (Integer ID of the record)
- Column-Stores
  - Fetch only required data
  - Built for attribute focused access
  - Utilize todays hardware better
  - Allow efficient column-wise light weight compression
  - In case disk-based: tuple reconstruction and inserts way too slow

Summary (contd.)

- Mixed Workload Environments need
  - Fast tuple reconstruction
  - Single insert capability
  - Fast set processing
  - Fast full table scan option

⇨ Combination of **In-Memory** Data Processing and **Column-wise** Data Organization

In-Memory Databases

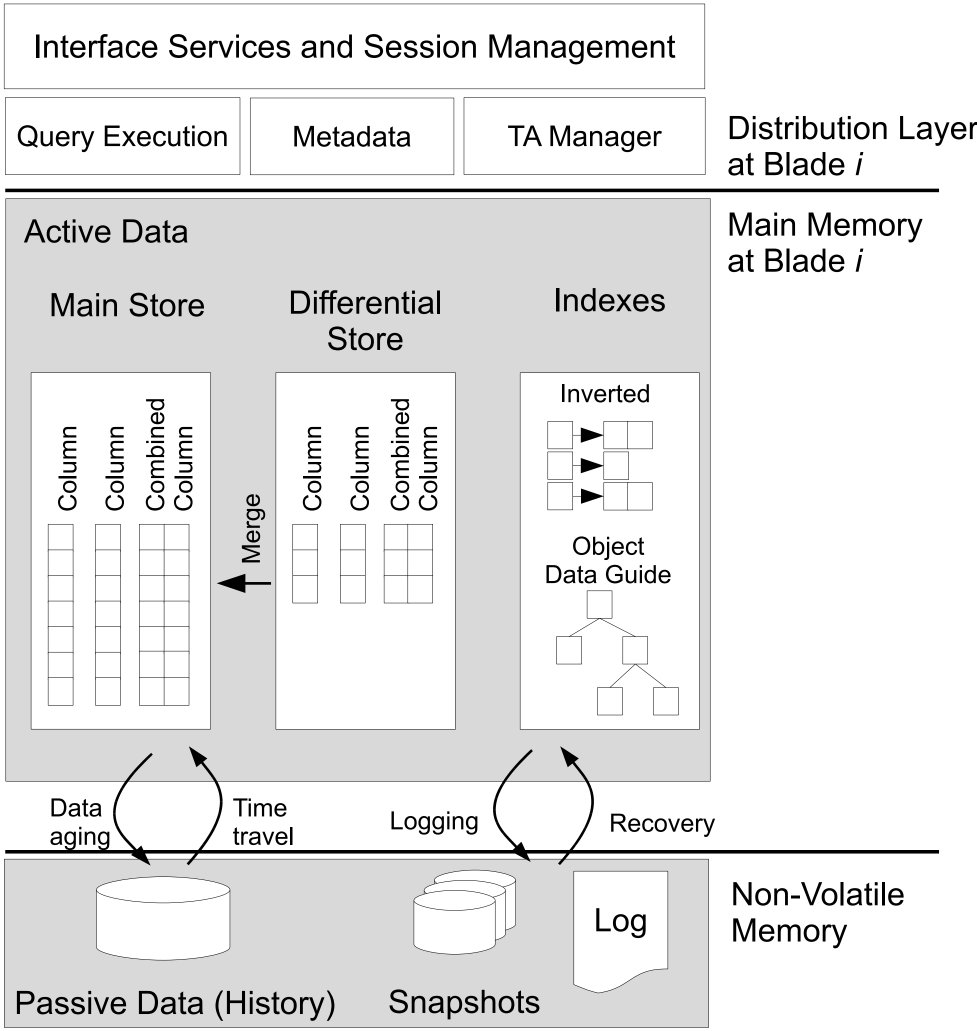
Jens Krueger

Enterprise Platform and Integration Concepts
Hasso Plattner Intitute

In-Memory Database

In-Memory Database (IMDB)

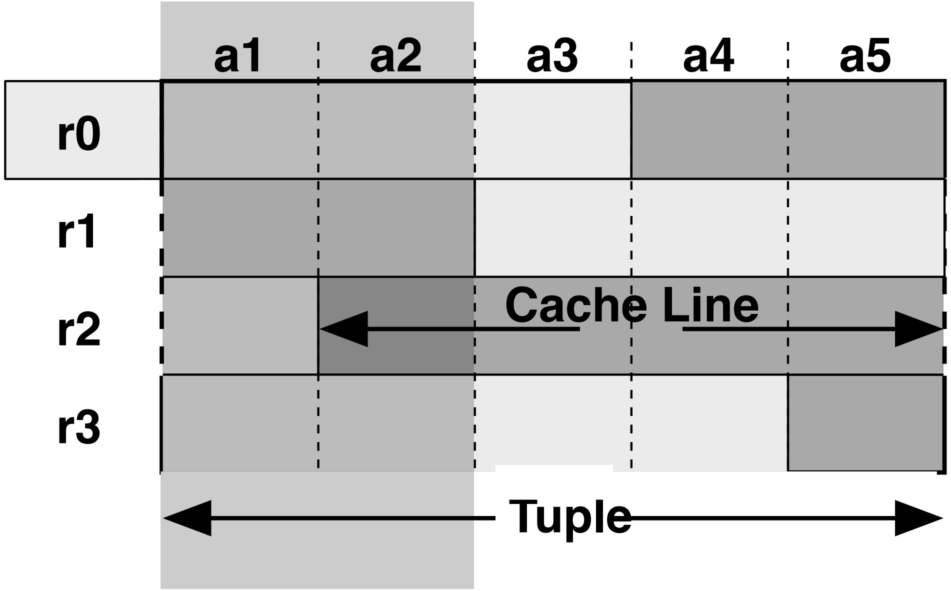
- Data resides **permanently** in main memory
- Main Memory is the **primary** *"persistence"*
- Still: logging to **disk**/recovery from **disk**
- Main memory access is the new **bottleneck**
- Cache-conscious algorithms/ data structures are **crucial** (locality is king)



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



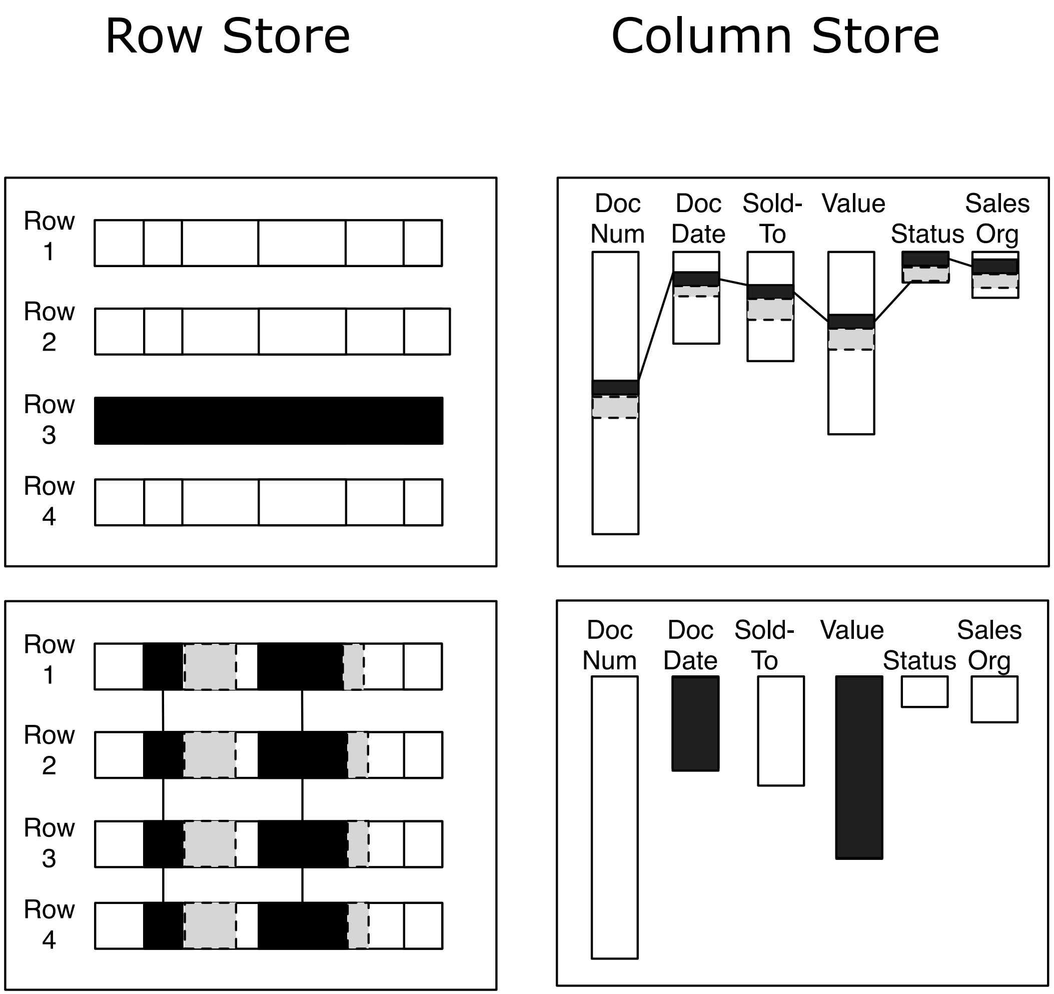
How to optimize an IMDB?

- Exploit sequential access, leverage locality

  -> Column store
- Reduce I/O
  - Compression
- Direct value access

  -> Fixed-length (compression schemes)
- Late Materialization
- Parallelize (presentation tomorrow..)

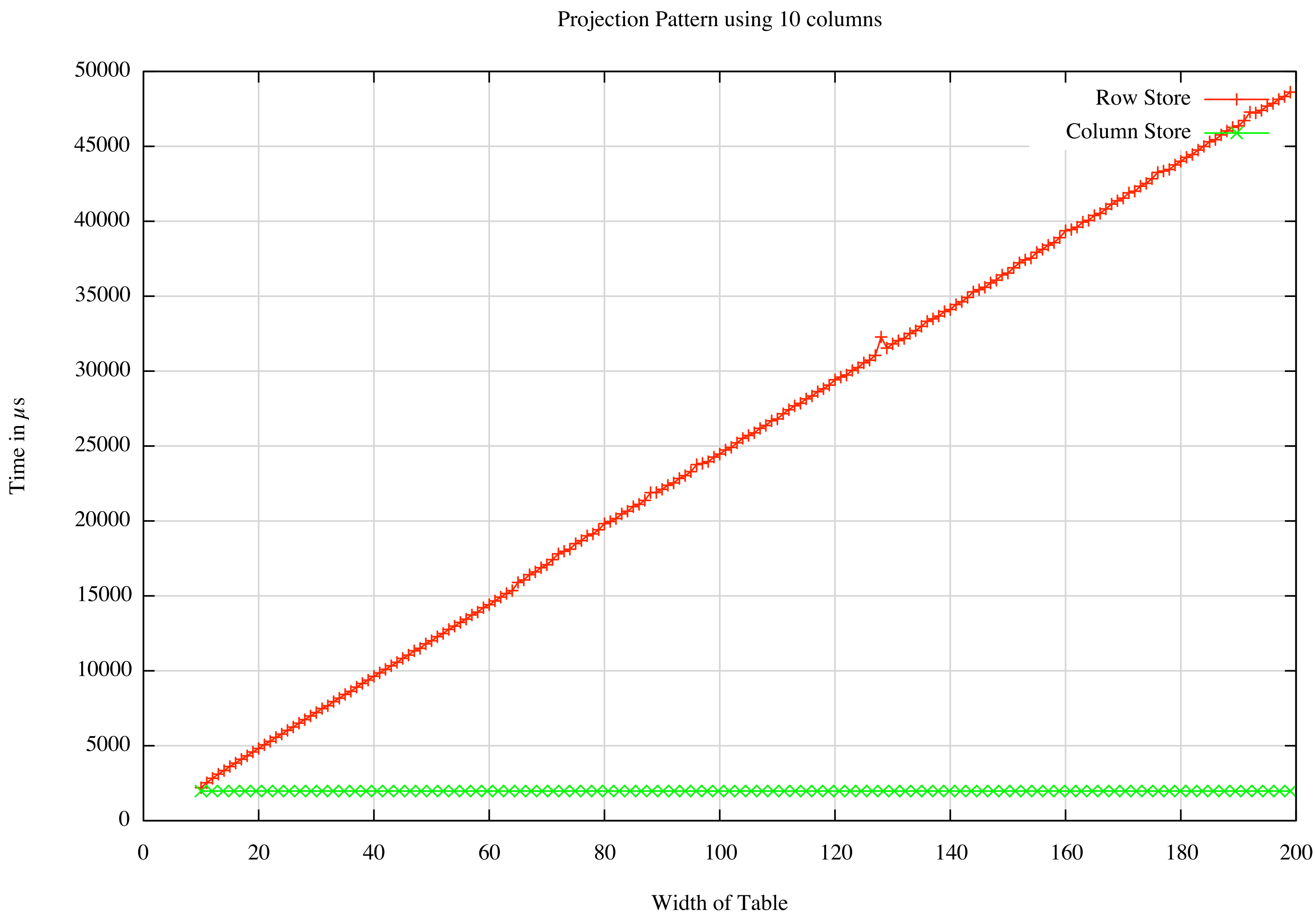
Row- or Column-oriented Storage



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

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

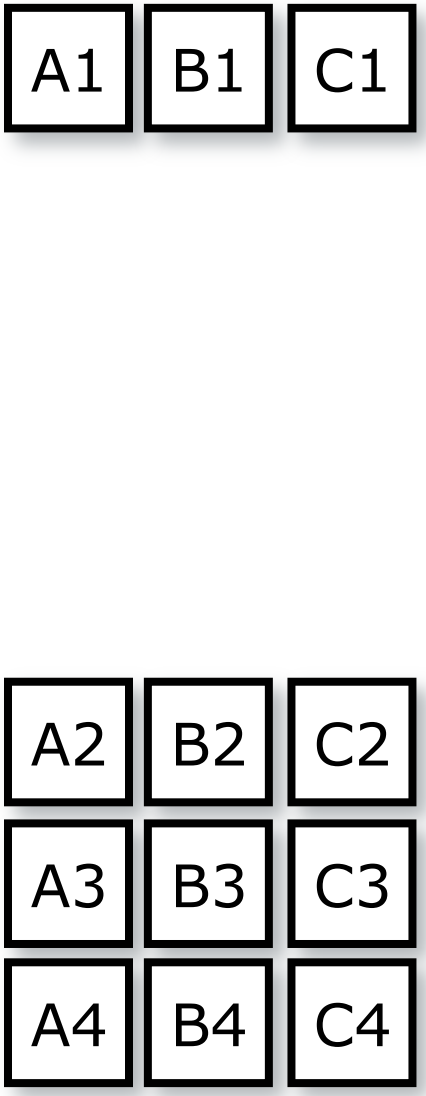
Projection of 10 Columns in HYRISE



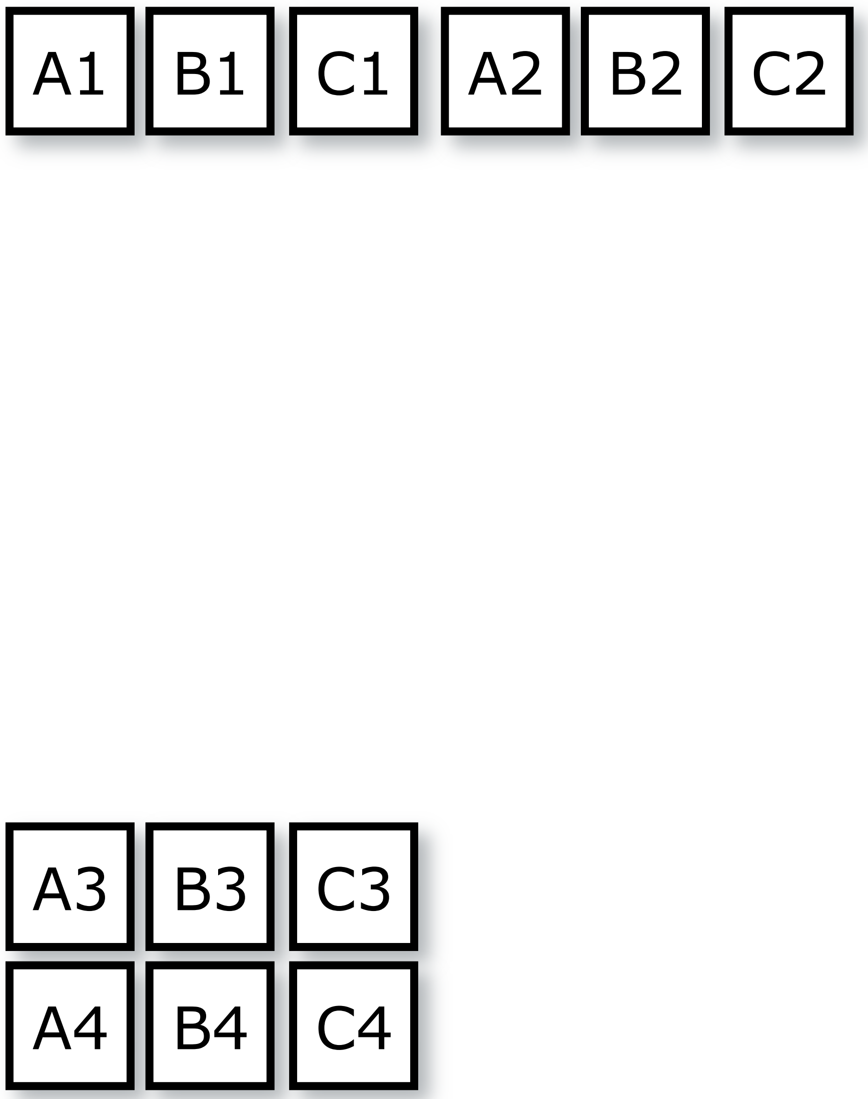
Row-oriented storage

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

Row-oriented storage



Row-oriented storage



Row-oriented storage

A1	B1	C1	A2	B2	C2	A3	B3	C3

A4	B4	C4

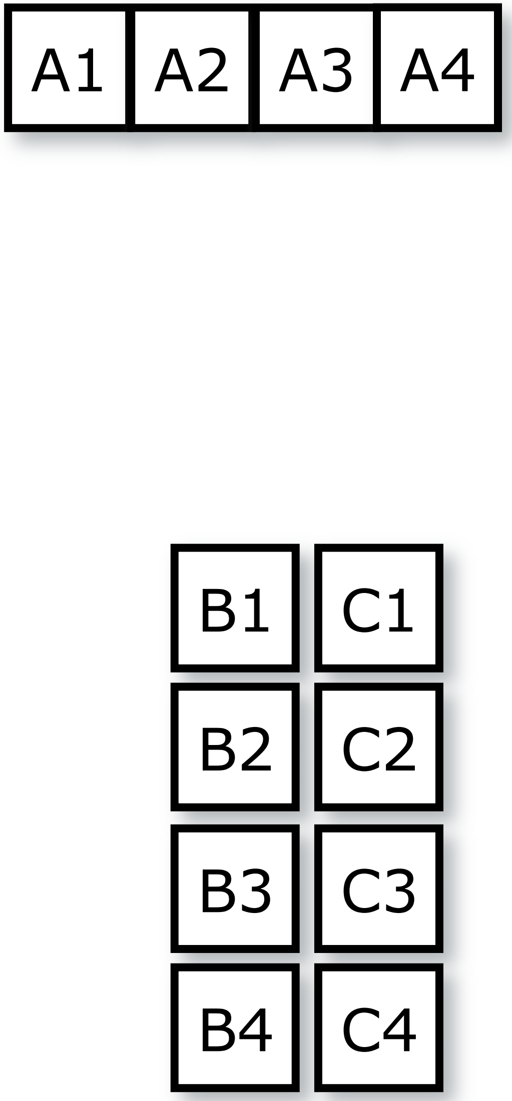
Row-oriented storage

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

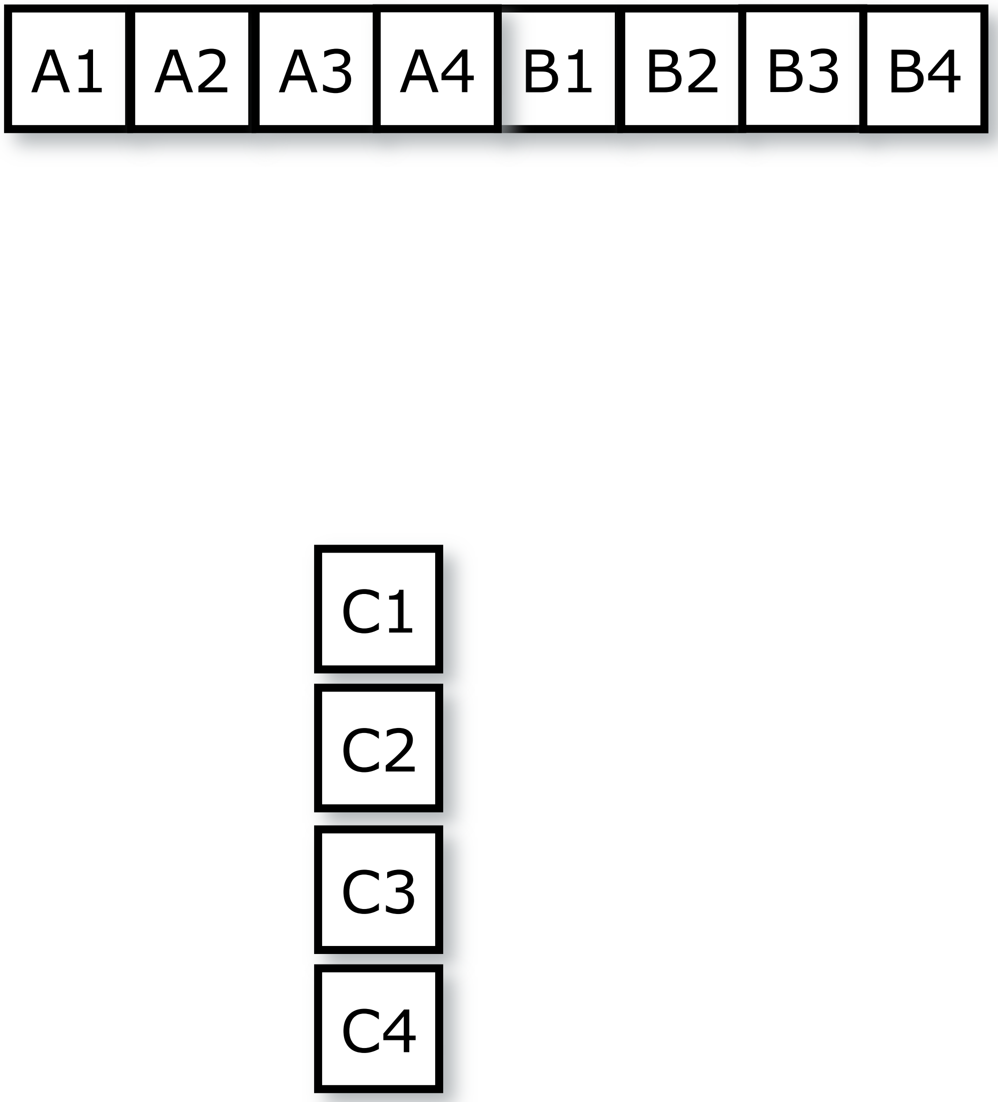
Column-oriented storage

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

Column-oriented storage



Column-oriented storage

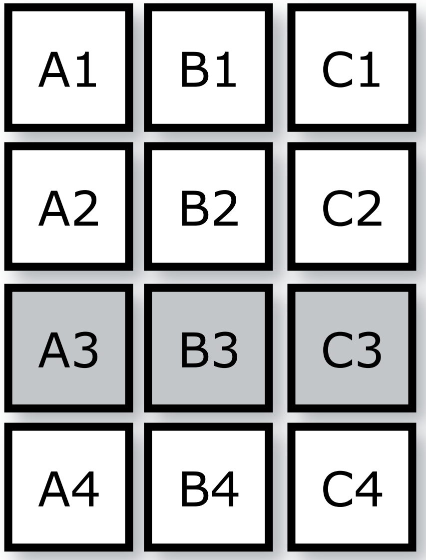


Column-oriented storage

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

Example: OLTP-Style Query

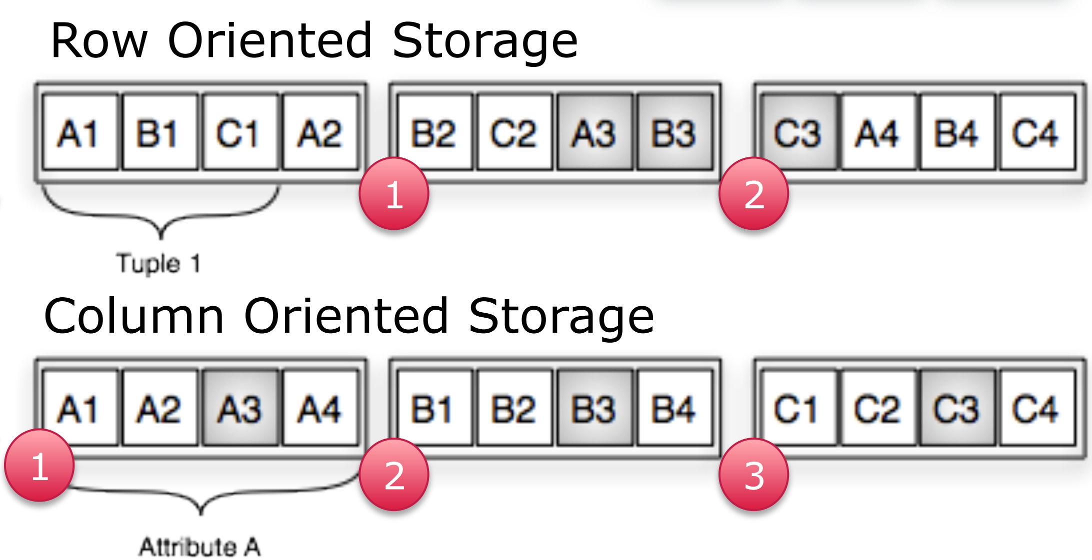
```
struct Tuple {
int a,b,c;
};

Tuple data[4];
fill(data);

Tuple third = data[3];
```



Example: OLTP-Style Query

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

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

Tuple data[4];
fill(data);

Tuple third = data[3];
```



Cache line

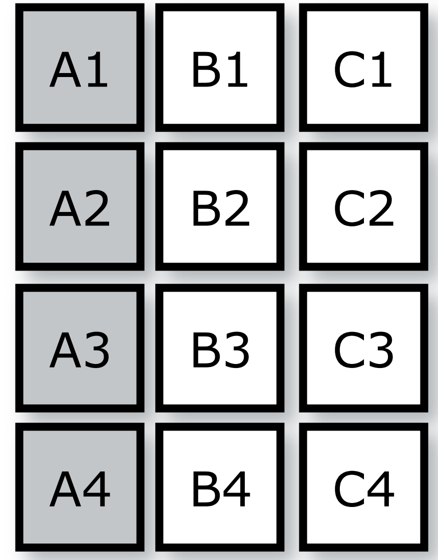
Example: OLAP-Style Query

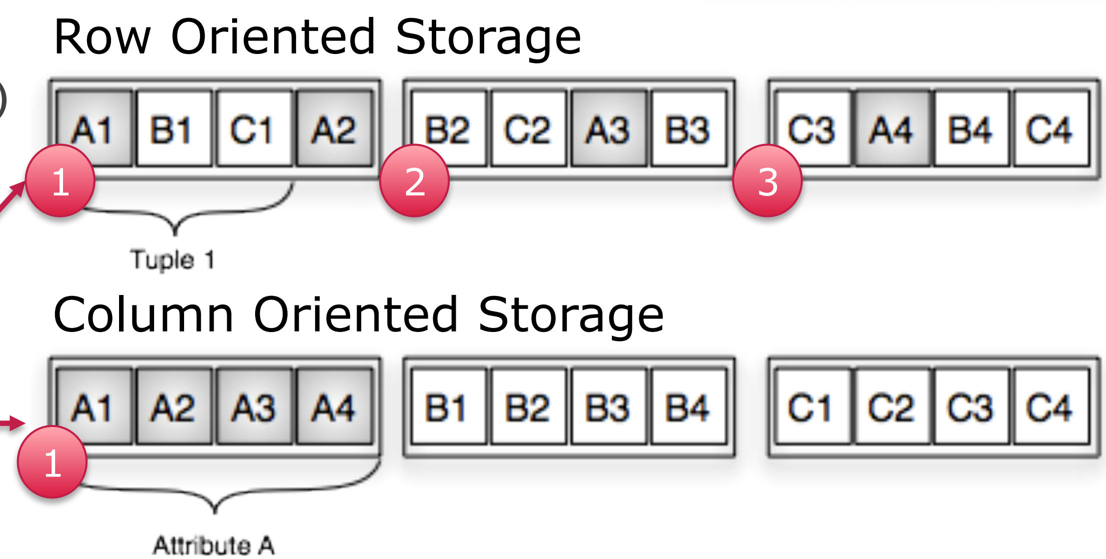
```
struct Tuple {
int a,b,c;
};

Tuple data[4];
fill(data);

int sum = 0;

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

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

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

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;
```



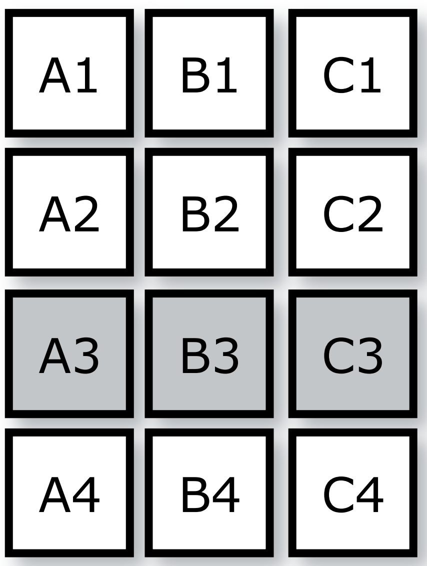


Cache line

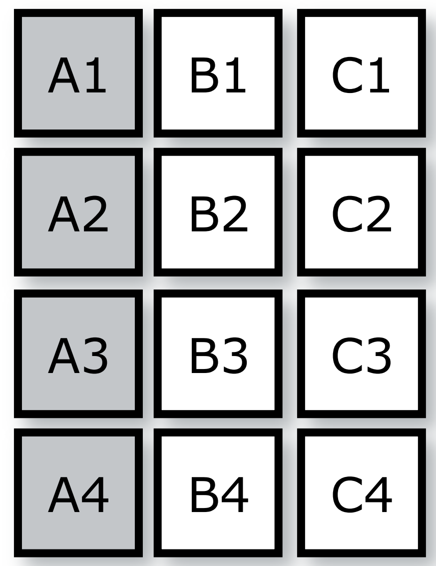
Mixed Workloads

- Mixed Workloads involve attribute- and entity-focused queries

OLTP-style queries



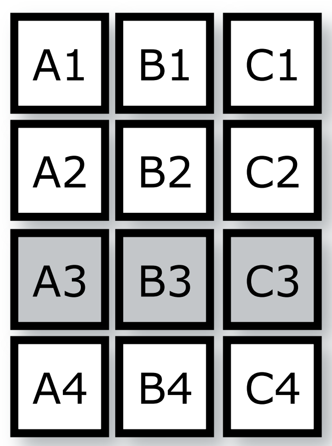
OLAP-style queries



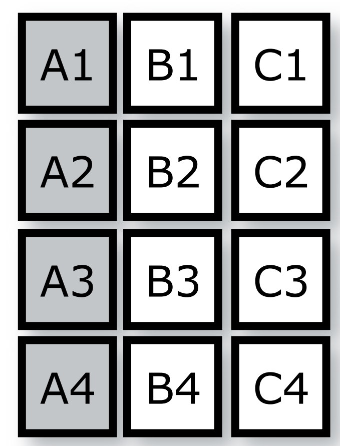
Mixed Workloads: Choosing the Layout

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

OLTP-style queries



OLAP-style queries



Compression in In-Memory Databases

Jens Krueger

Enterprise Platform and Integration Concepts
Hasso Plattner Intitute

Motivation

- Main memory access is the bottleneck
- Idea: Trade CPU time to compress and decompress data
- Lightweight Compression
  - **Lossless**
  - **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

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

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 (valueID) for each distinct value
- Store valueID instead of value in attribute vector
- Enables offsetting with bit-encoded fixed-length data types

Table

RecId 1	JAN	INTEL	€1
RecId 2	FEB	ABB	€2
RecId 3	MAR	HP	€2
RecId 4	APR	INTEL	€4
RecId 5	MAY	IBM	€5
RecId 6	JUN	IBM	€5
RecId 7	JUL	SIEMENS	€4
RecId 8	AUG	INTEL	€3
…	...	...	...

Attribute: Company Name

Attribute Vector

RecId	ValueId
1	4
2	1
3	2
4	4
5	3
6	3
7	5
8	4

Dictionary

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

Inverted Index

ValueId	RecIdList
1	2
2	3
3	5,6
4	1,4,8
5	7

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 10.000 distinct values
  - Each: 1 key, 1 value => 32 Bytes
  - ~ 0.3 Megabytes
  - 1 million * 4 Bytes = ~ 4 Megabytes
- Overall: ~4.3 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 - log2(distinct values)
  - Optimal for only a few distinct values
  - Re-encoding if more bits to encode needed
- Query execution (e.g. lookup & rage queries)
  - Use order-preserving dictionaries
  - ValueID’s have same order as uncompressed values
  - value1 $<$ value2 $<=>$ valueID1 $<$ valueID2

Materialization During Query Execution in Column Stores

Jens Krueger

Enterprise Platform and Integration Concepts
Hasso Plattner Intitute

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:

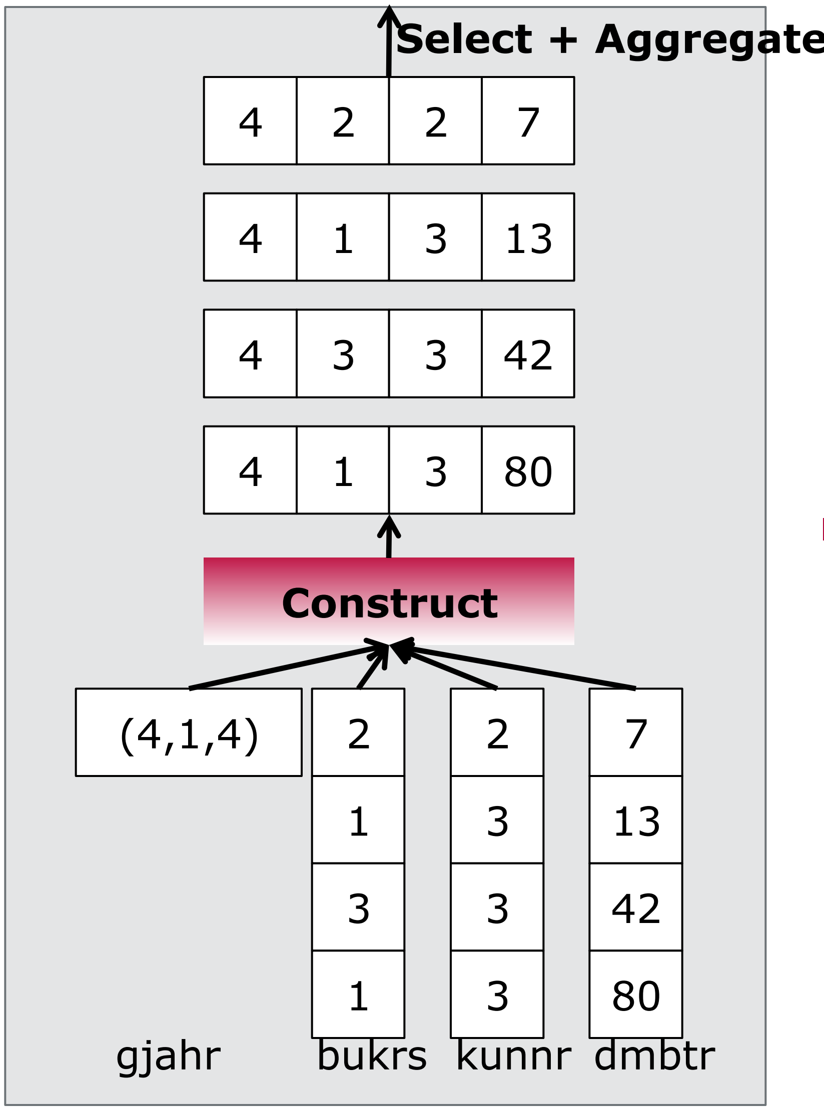
Query:

```
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

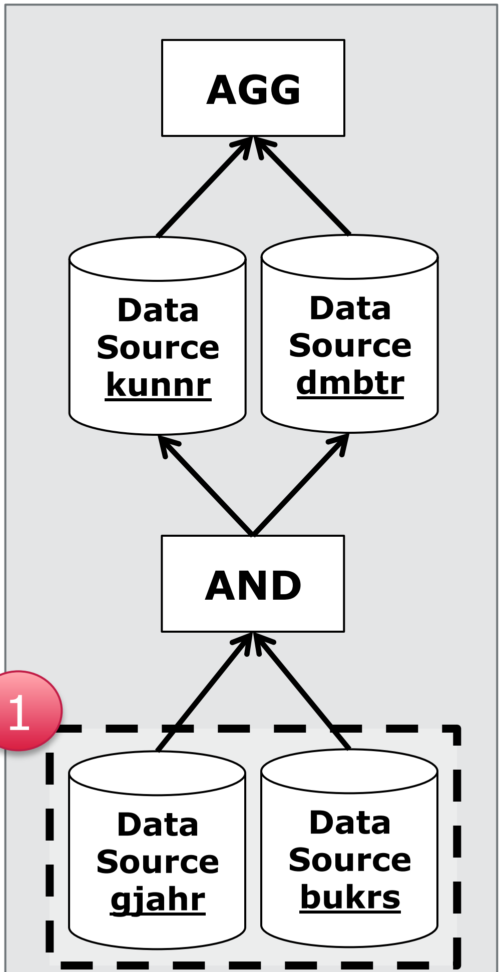
Query:

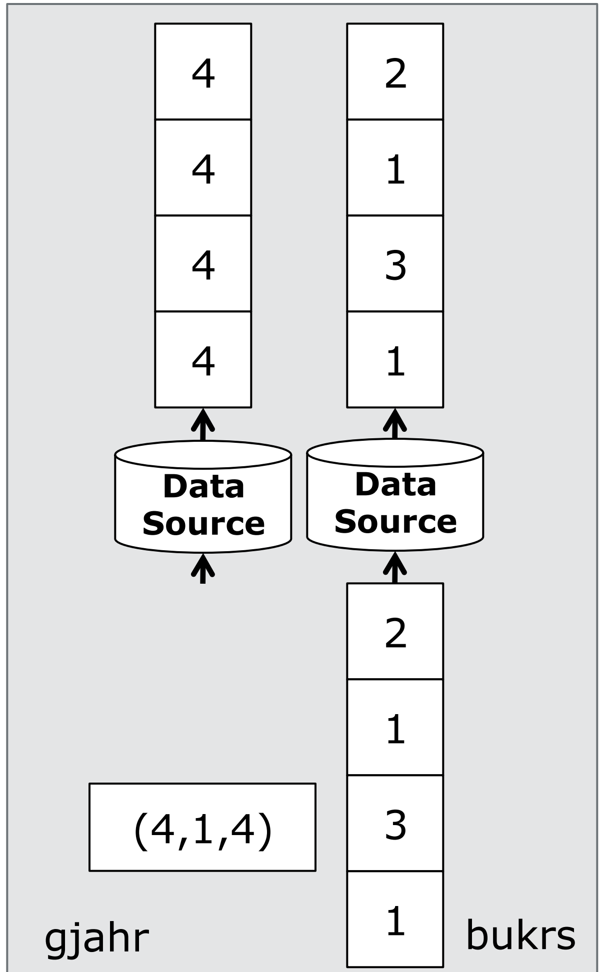
SELECT kunnr, sum(dmbtr)
FROM BSEG
WHERE gjahr = 4
AND bukrs = 1
GROUP BY kunnr

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

Late materialization I

- **Operate on columns**





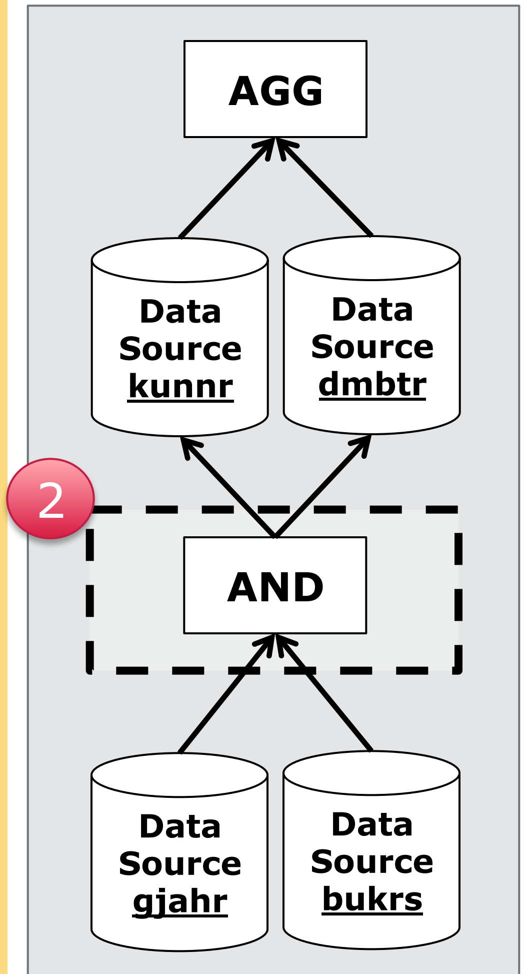
Query:

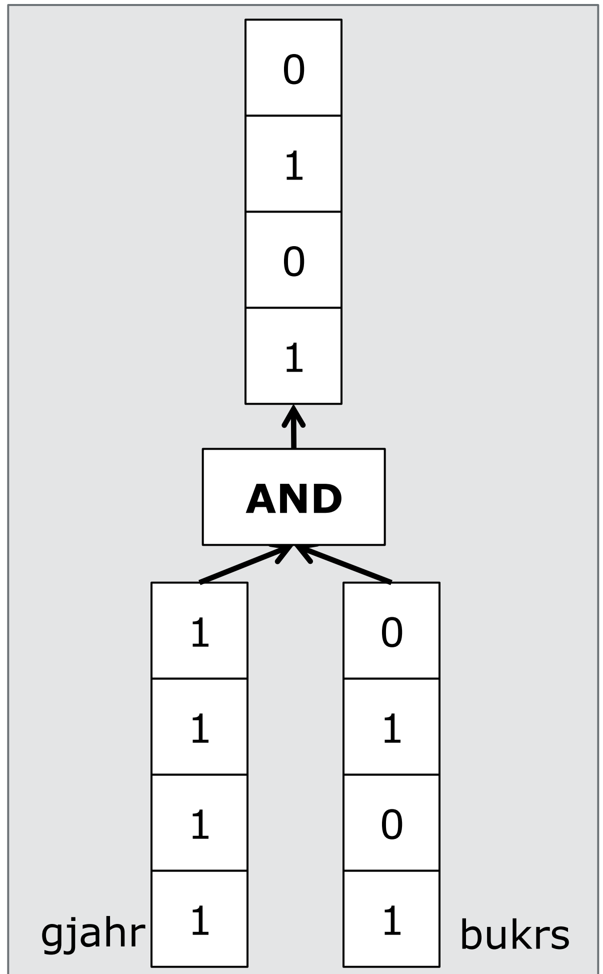
SELECT kunnr, sum(dmbtr)
FROM BSEG
WHERE gjahr = 4
AND bukrs = 1
GROUP BY kunnr

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

Late materialization II

- **Operate on columns**





Query:

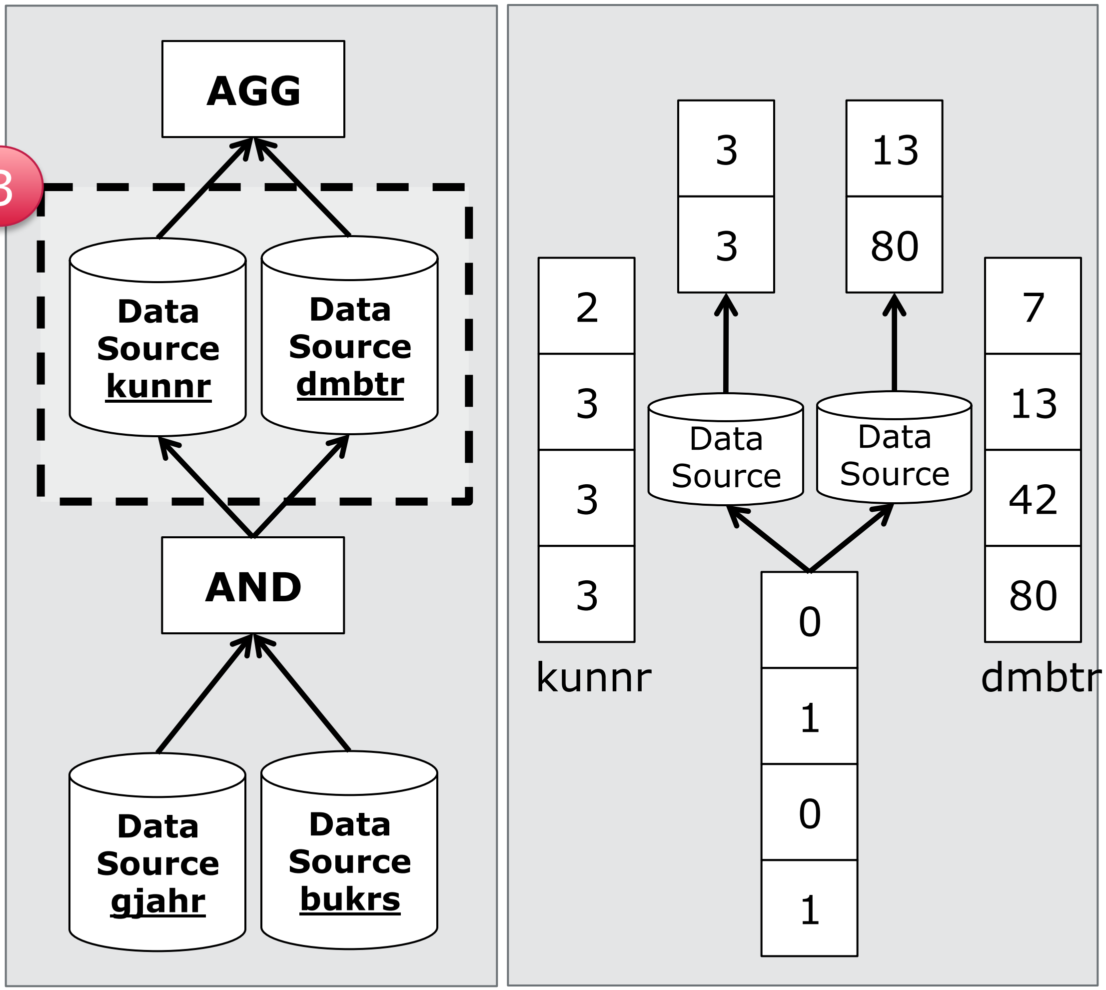
SELECT kunnr, sum(dmbtr)
FROM BSEG
WHERE gjahr = 4
AND bukrs = 1
GROUP BY kunnr

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

Late materialization III

- **Operate on columns**



Query:

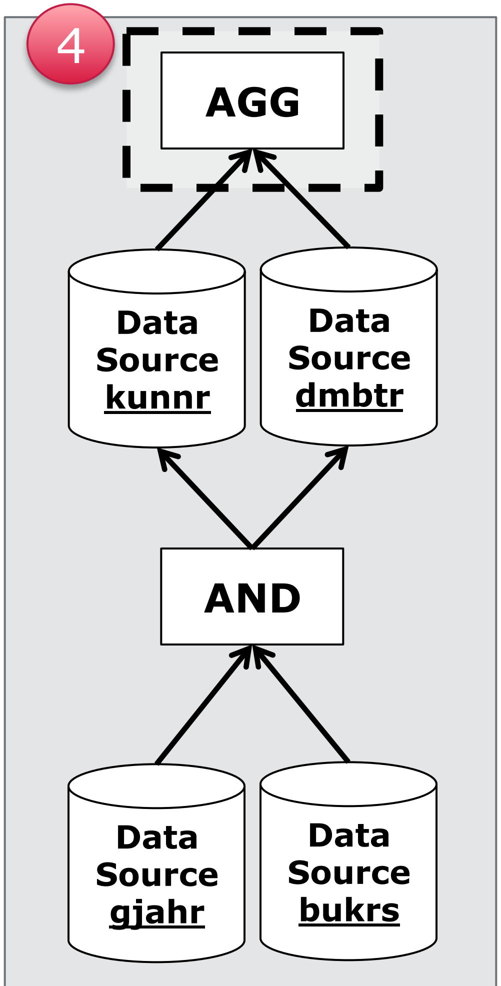
SELECT kunnr, sum(dmbtr)
FROM BSEG
WHERE gjahr = 4
AND bukrs = 1
GROUP BY kunnr

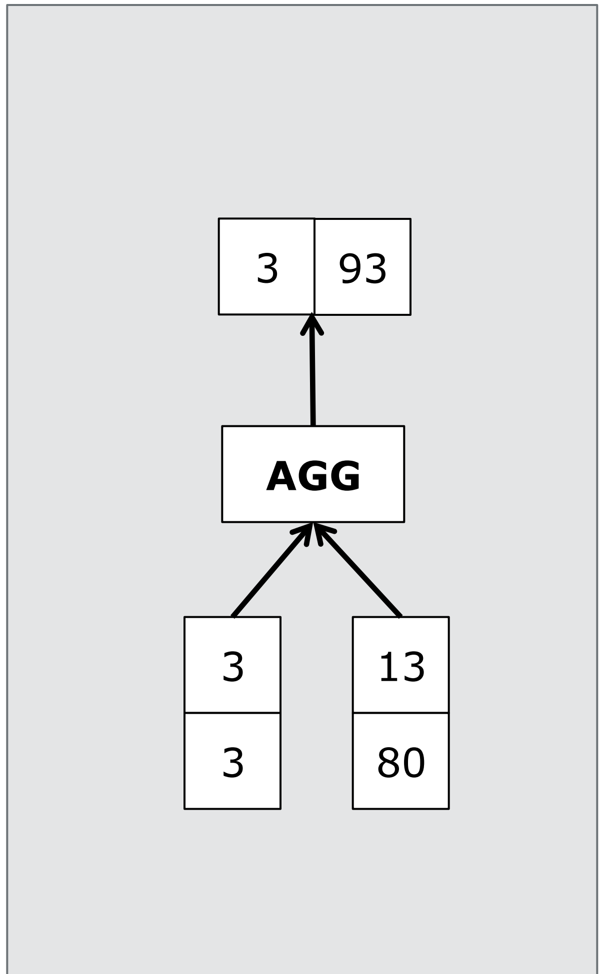
4	2	2	7
4	1	3	13
4	3	3	42
4	1	3	80
gjahr	bukrs	kunnr	dmbtr

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

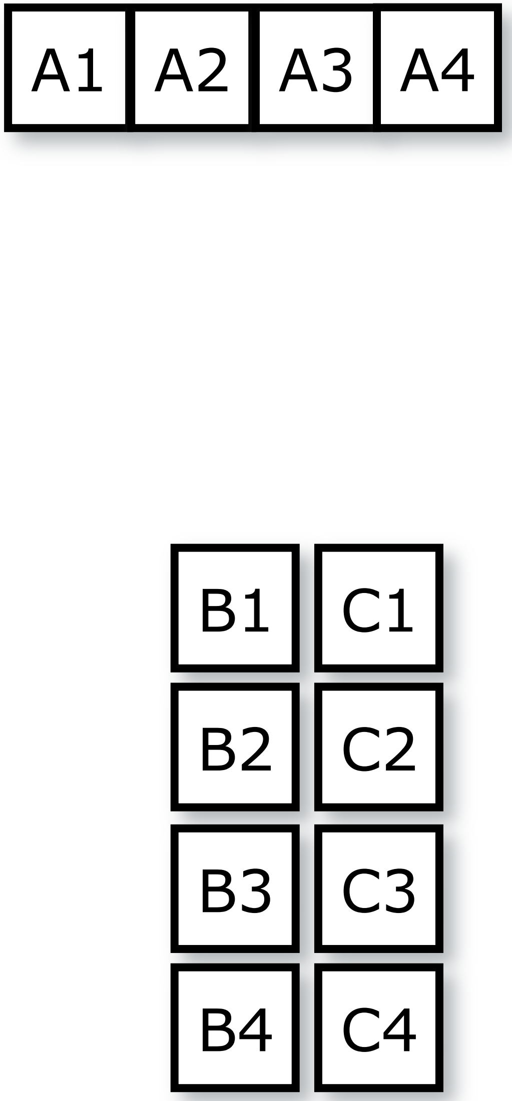
Backup

HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Hybrid: Grouping of Columns

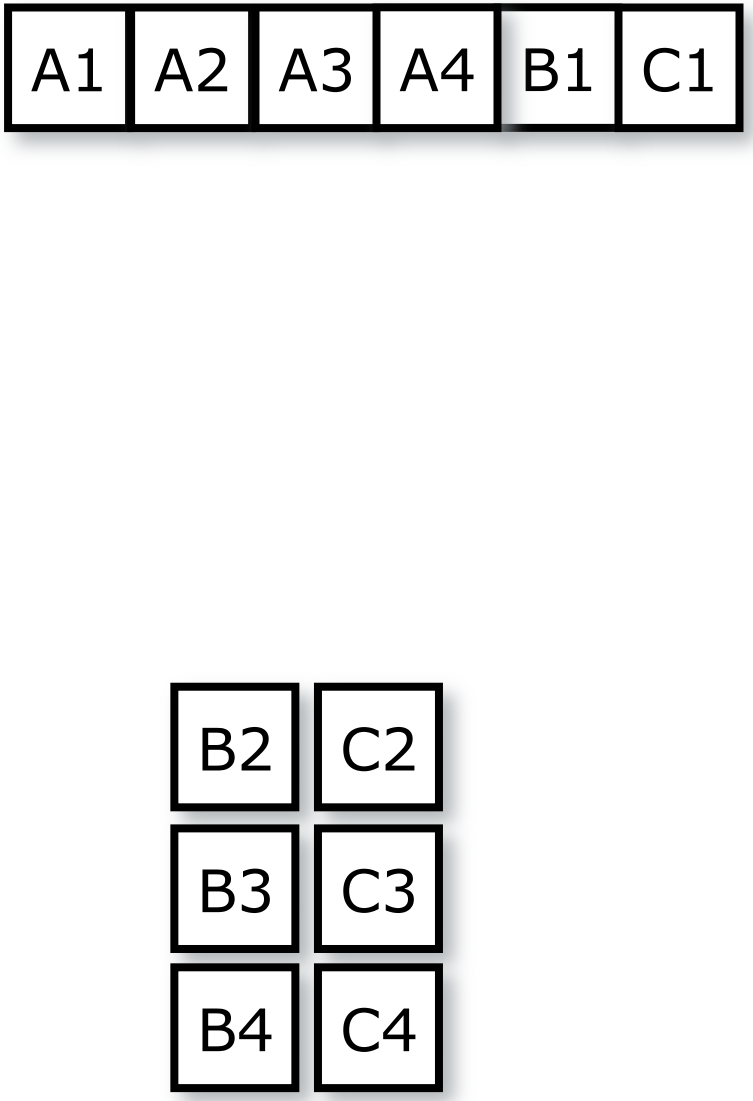
A1	B1	C1
A2	B2	C2
A3	B3	C3
A4	B4	C4

Hybrid: Grouping of Columns

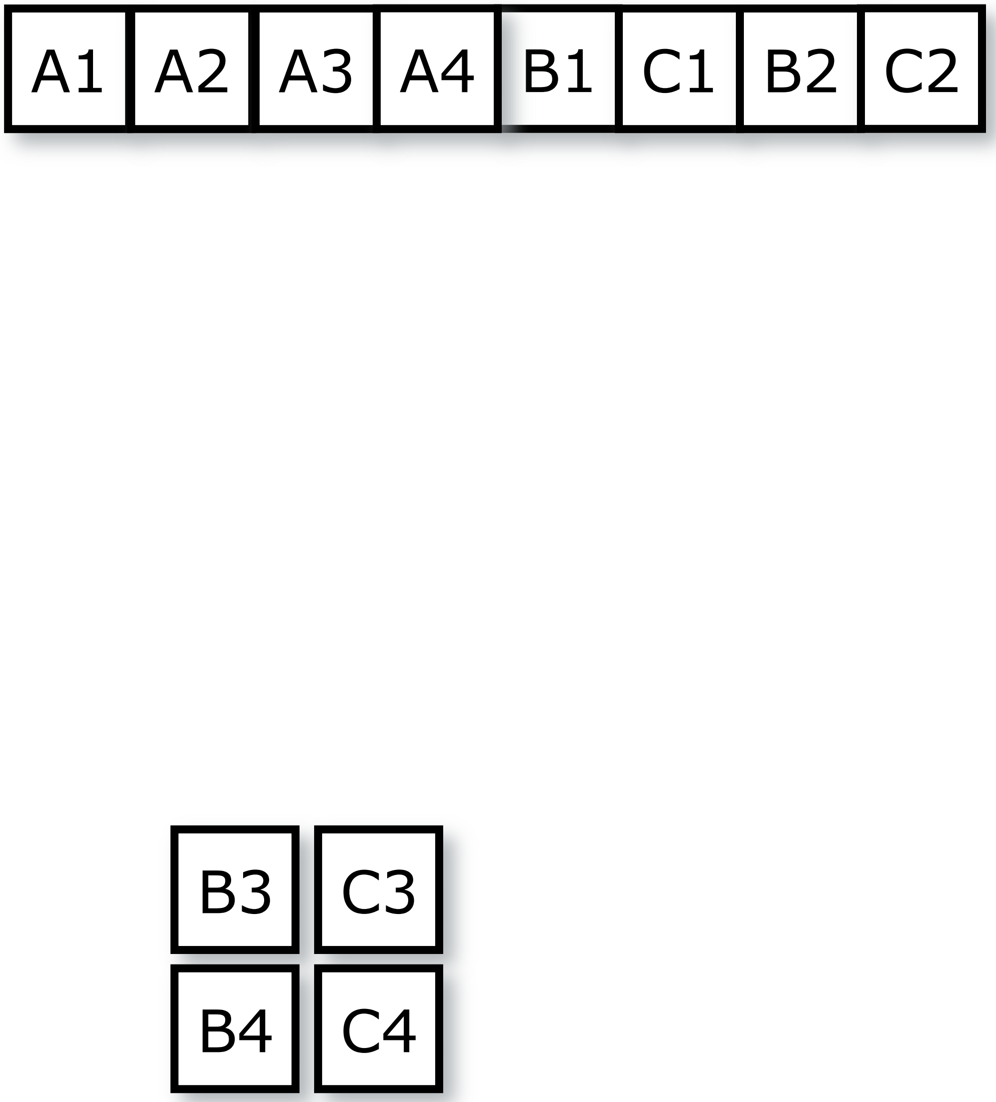


HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

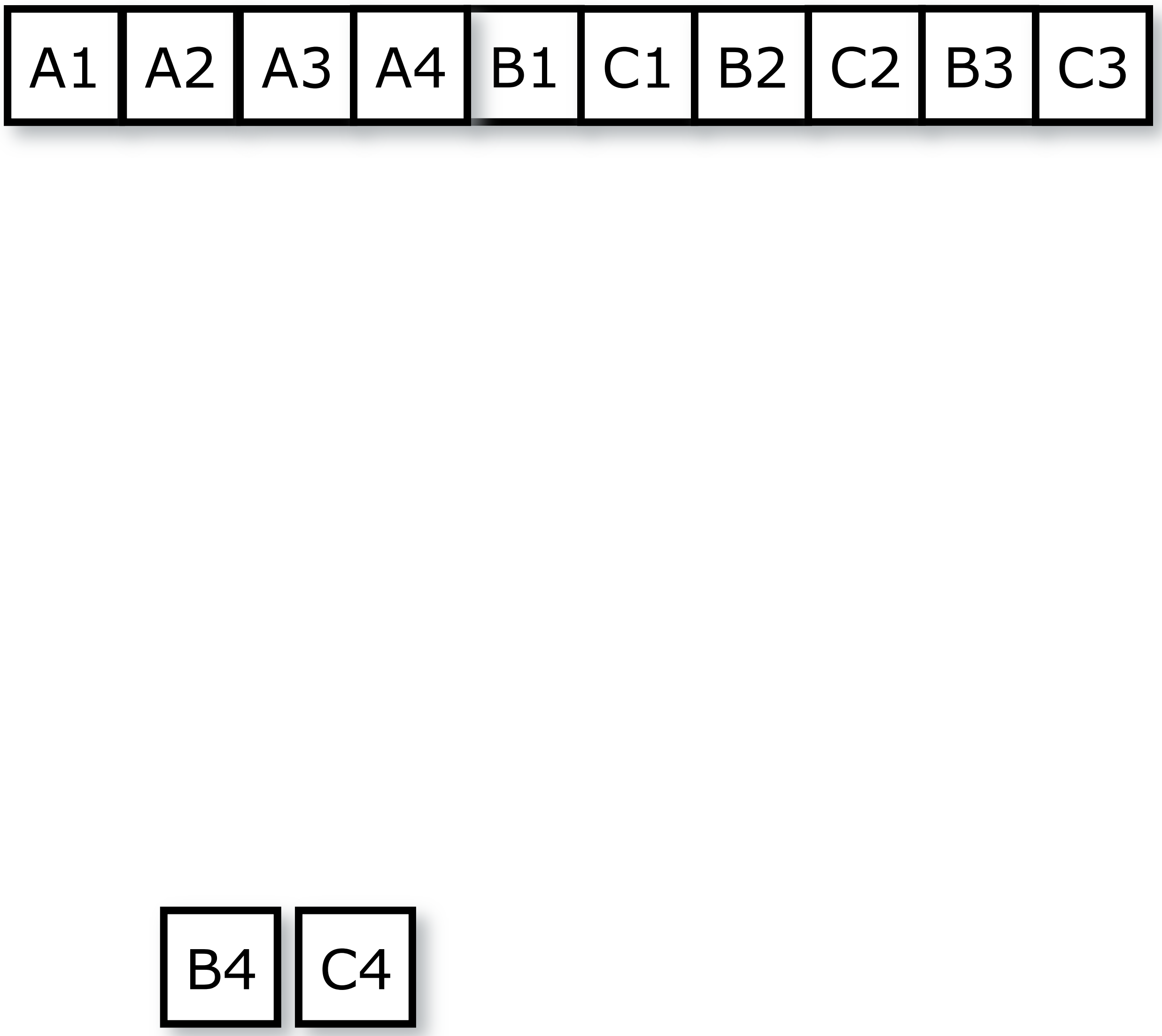
Hybrid: Grouping of Columns



Hybrid: Grouping of Columns



Hybrid: Grouping of Columns

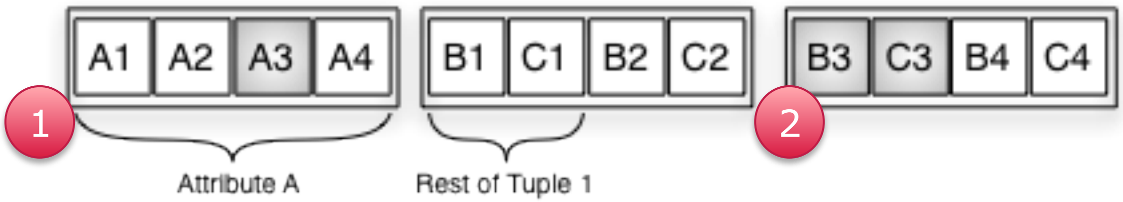


Hybrid: Grouping of Columns

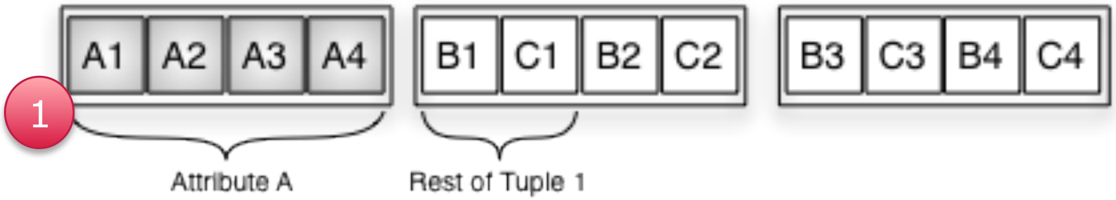
A1	A2	A3	A4	B1	C1	B2	C2	B3	C3	B4	C4

Hybrid: Grouping of Columns

Access tuple 3



Query attribute A



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