In-Memory Databases

Jens Krueger

Recap

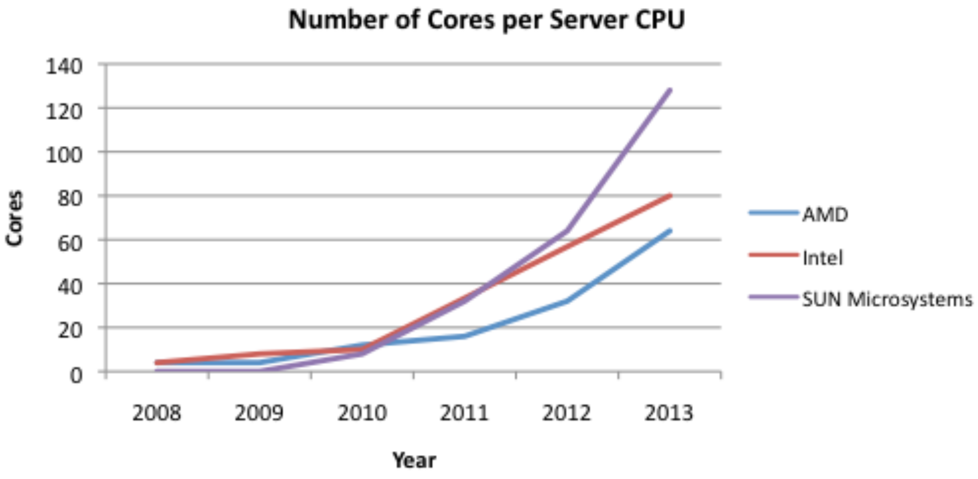
Recap: Workload Characteristics

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

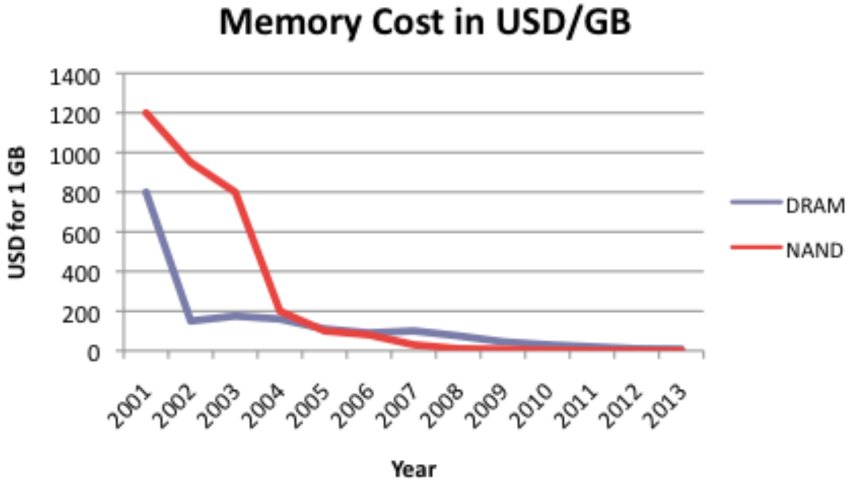
- Clark D. French, „Teaching an OLTP Database Kernel Advanced Datawarehousing Techniques” ICDE 97

Recap: Hardware Trends

- Multi-Core Technology
  - Moore’s Law: “…number of transistors … doubling approximately every 18 month”
  - CPU frequency hit limit in 2002, but Moore’s law holds today



- Main-Memory Technology
  - Increased size: up to 1TB of main-memory on one main board in 2010
  - Constantly dropping costs
  - RAM vs. disk access time: 100 ns vs. 10.000.000 ns



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

Memory Access

Data Processing

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

- Not CPU bound
- **But** bandwidth bound
- Gets even worse with multi-cores

⇨ CPU can process data faster than it can read it

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

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**

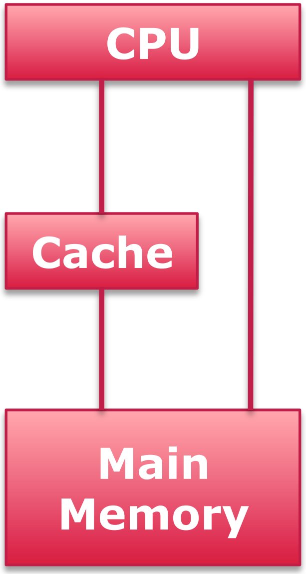
  - Fast but small memory on the top
  - Slow but lots of memory at the bottom

	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

Memory Basics II

- **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**

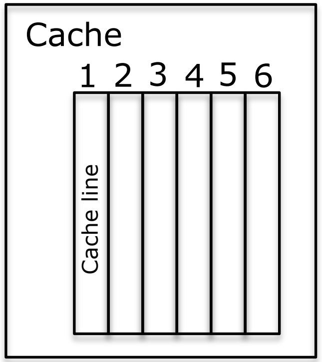
  Data was **not** found in the cache. CPU has to load data from main memory into cache (**miss penalty**).

Memory Basics III

- **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

Cache line replacement

Eviction of cache lines is needed

- Strategies for replacement (hardware driven)
  - Least recently used
    - Least accessed line is replaced
    - Assumption: least likely to access accessed
    - Expensive maintenance
  - Random
    - Random line eviction
    - Easy to implement

Write data

Reads dominate cache access but what about writes?

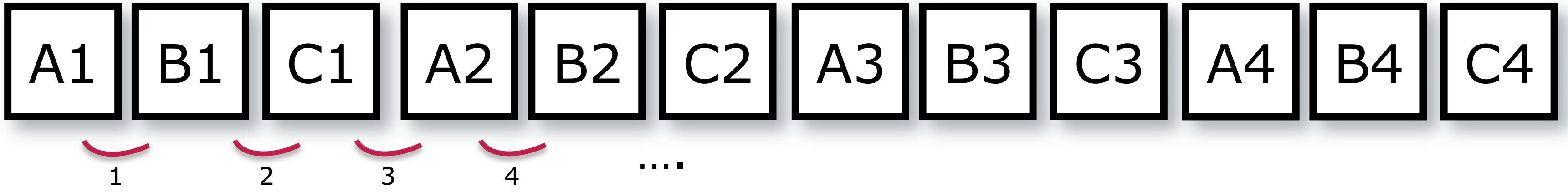
Write through

- Data is written to cache and main memory at the same time
- Maintains memory consistency
- As slow as low-level memory access

Write back

- Write back to cache only
- Dirty flag is used
- While evicted dirty blocks/lines are written back to main memory
- Consistency issues

Example

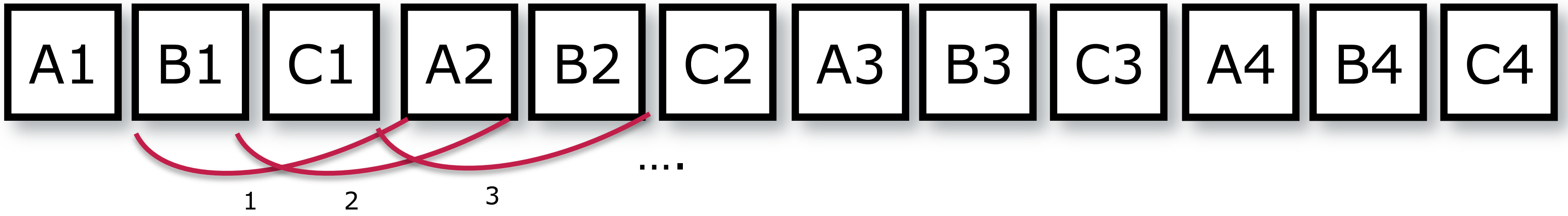


```
for (r = 0; r < rows; r++)
   for (c = 0; c < cols; c++)
     read[c] = table[r * cols + c];
```

Simulates sequential access

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

Example

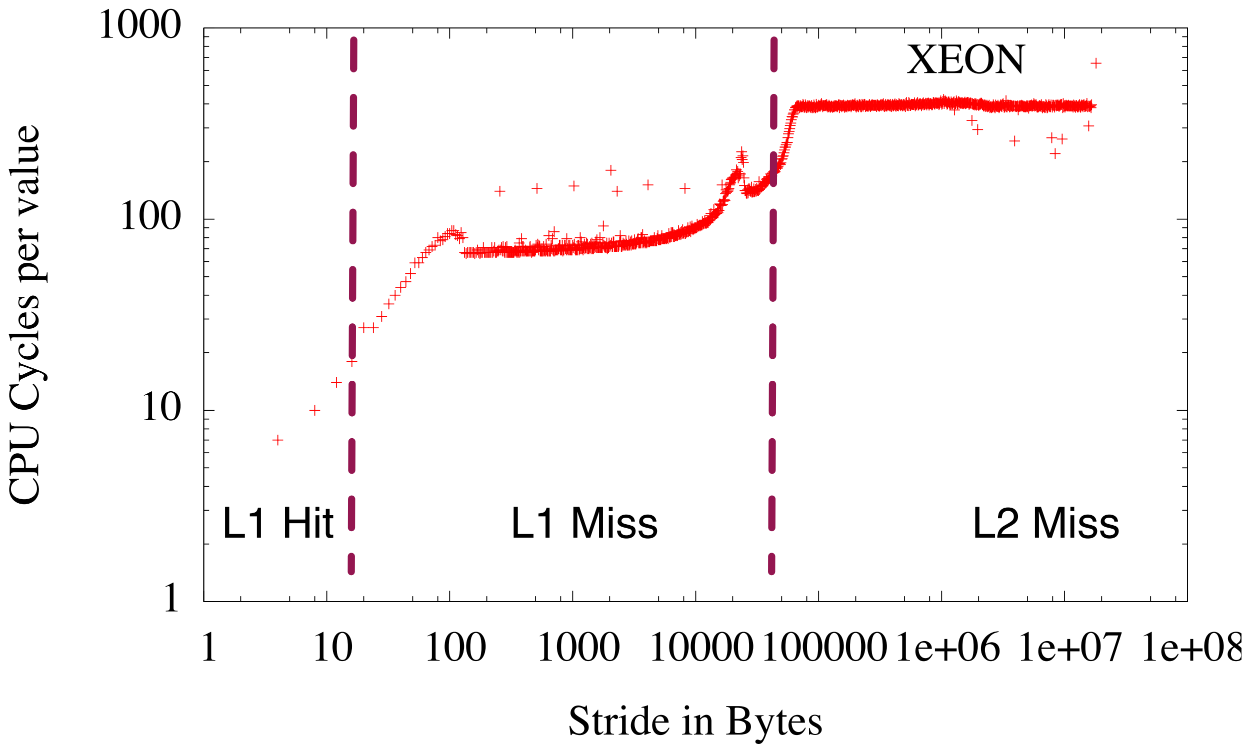


```
for (c = 0; c < cols; c++)
    for (r = 0; r < rows; r++)
      read[r] = table[c * cols + r];
```

Simulates traversal sequential access

- Fixed stride (access offset) leads to cache misses
- Varying stride allows to measure cache size

Evaluation


XEON
L1 Hit
L1 Miss
L2 Miss
CPU Cycles per value
1000
100
10
1
1
10
100
1000
10000
100000
1e+06
1e+07
1e+08
Stride in Bytes

In-Memory Database I

In a 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 is the new bottleneck
- Cache-conscious algorithms/data structures are crucial

Differences from disk-based systems

- Volatile
- Direct access
- Access time
- Access cost

In-Memory Databases II

Can an entire database fits 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?

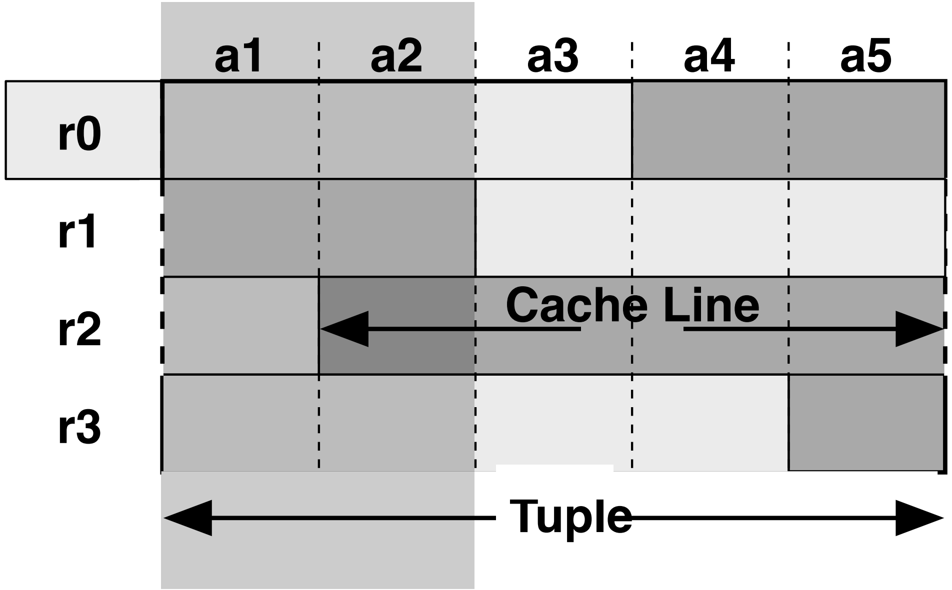
What is the difference between a IMDB and a disk-based DB with a large cache?

- 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.

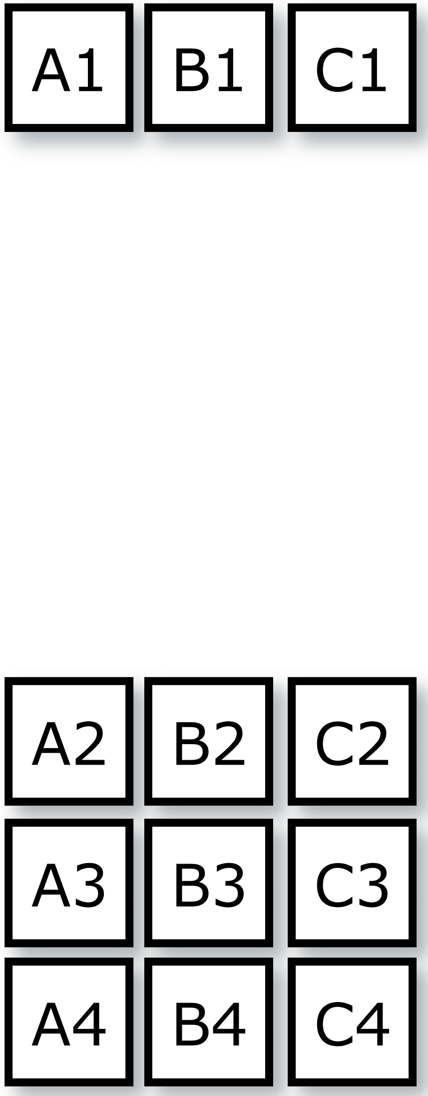
- Tuples are spanned over cache lines
- Wrong layout can lead to lots of (expensive) cache misses
- Row- or column-oriented can reduce cache misses if matching workload is applied



Row-oriented storage

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

Row-oriented storage



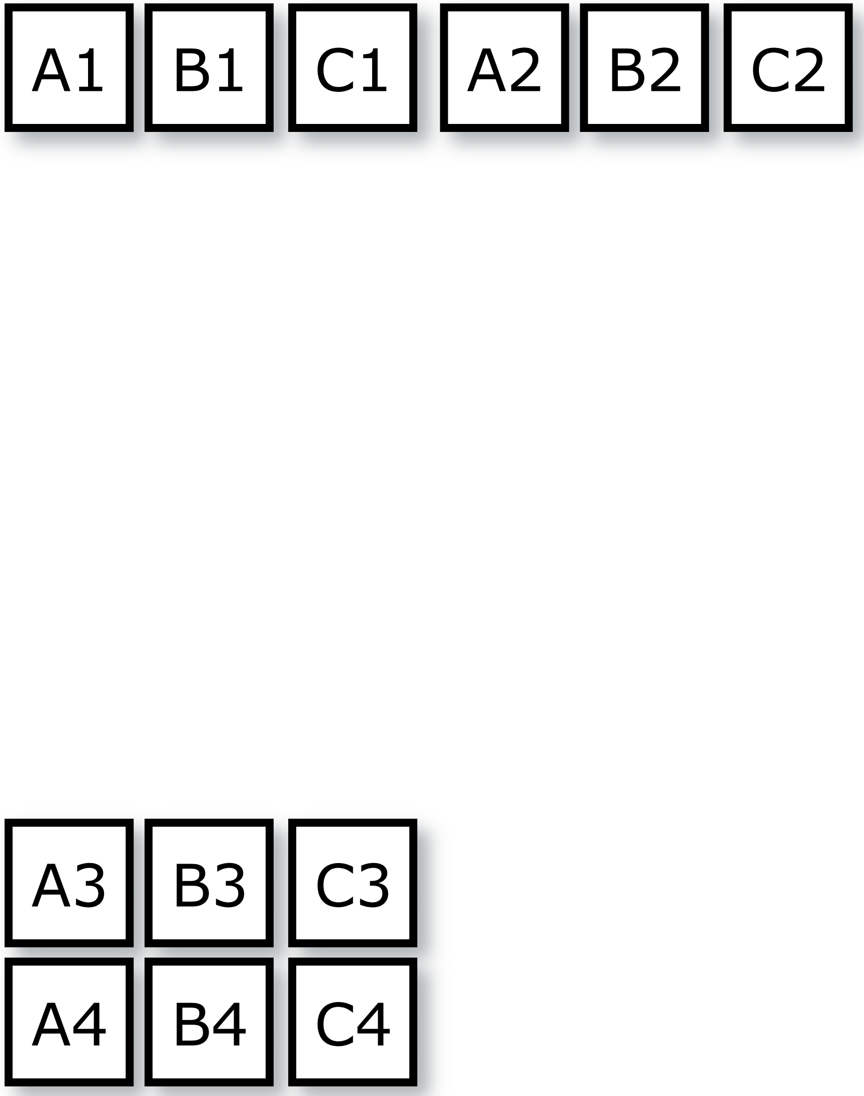
A1 B1 C1

A2 B2 C2

A3 B3 C3

A4 B4 C4

Row-oriented storage



A1 B1 C1 A2 B2 C2

A3 B3 C3

A4 B4 C4

Row-oriented storage

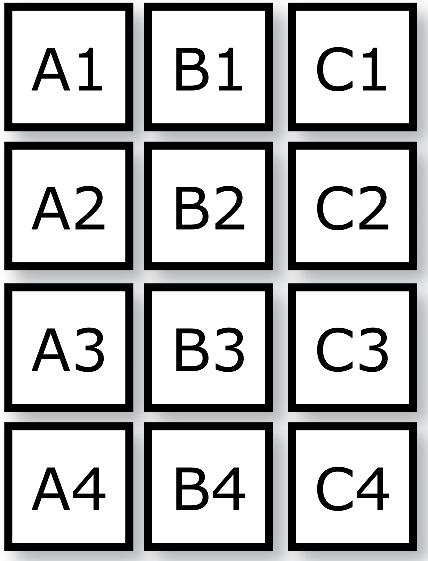
A1 B1 C1 A2 B2 C2 A3 B3 C3

A4 B4 C4

Row-oriented storage

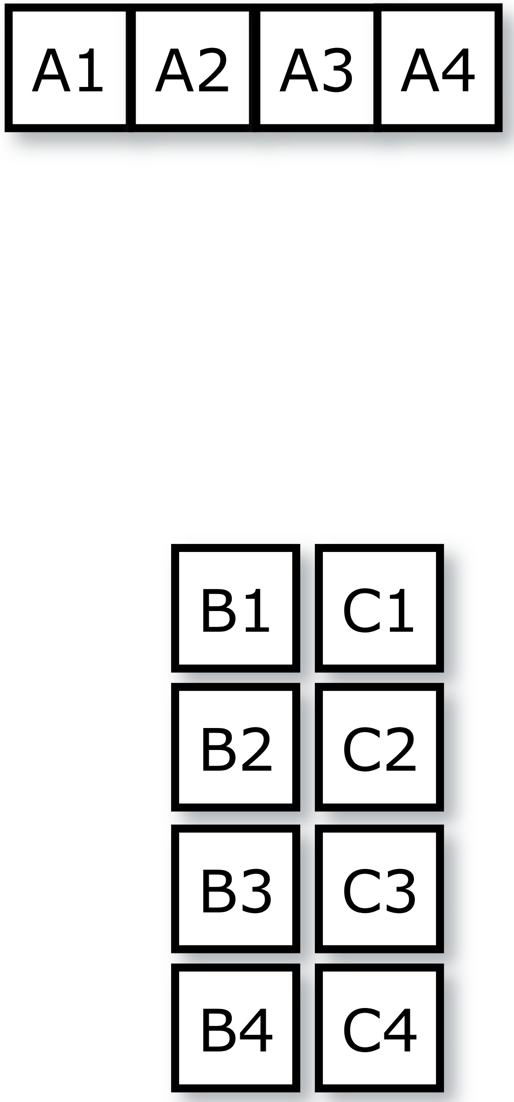
A1	B1	C1	A2	B2	C2	A3	B3	C3	A4	B4	C4

Column-oriented storage



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

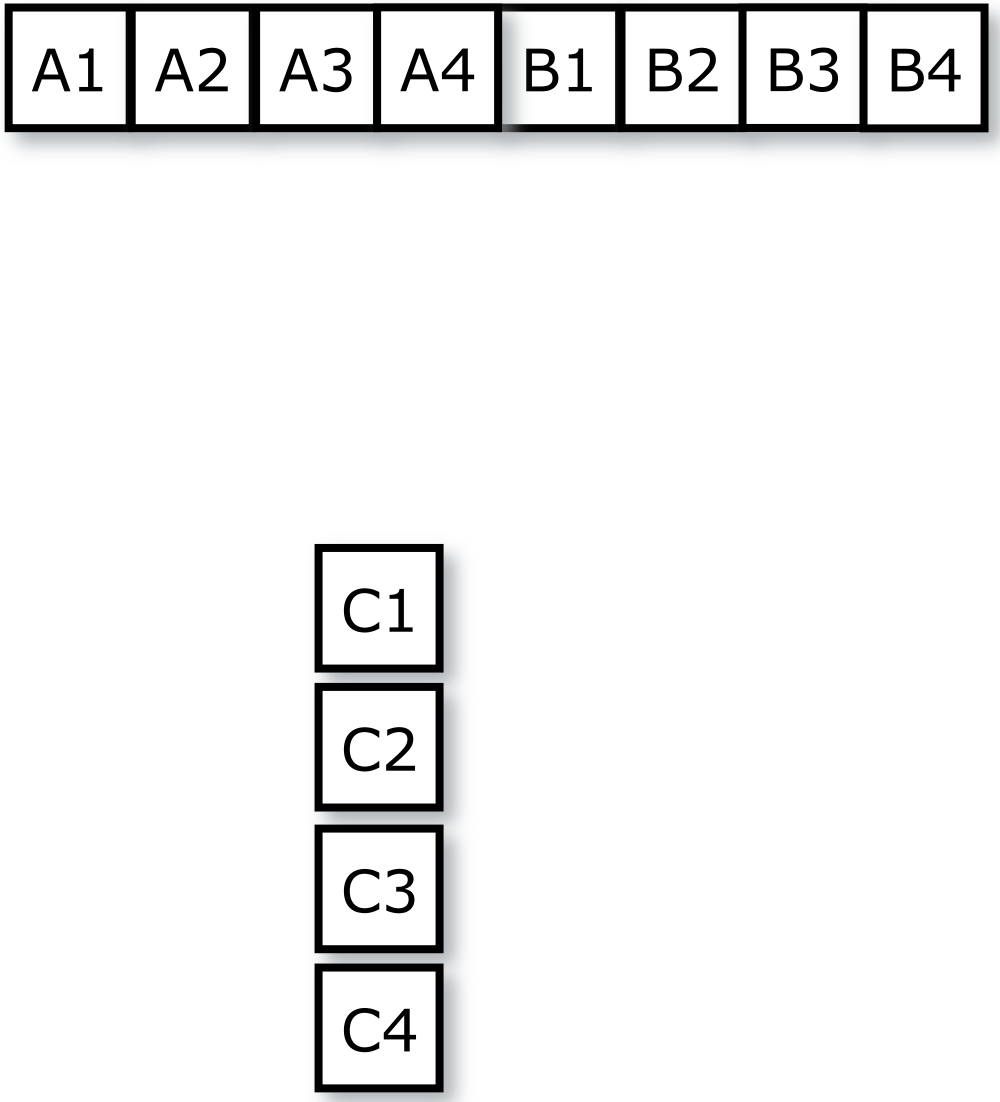
Column-oriented storage



A1	A2	A3	A4

B1	C1
B2	C2
B3	C3
B4	C4

Column-oriented storage



Column-oriented storage

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

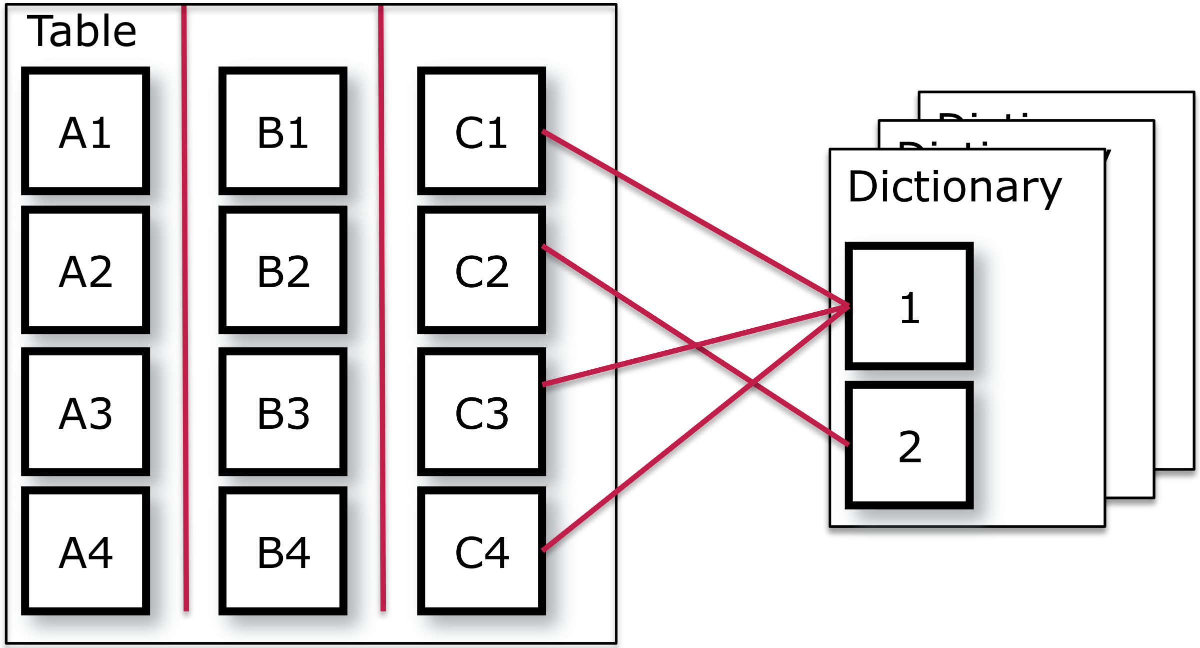
Column-oriented Storage

Pure vertical partitioning

- Table is decomposed into n arrays (n #of attributes)
- Arrays keep track of relations by position or separate ID

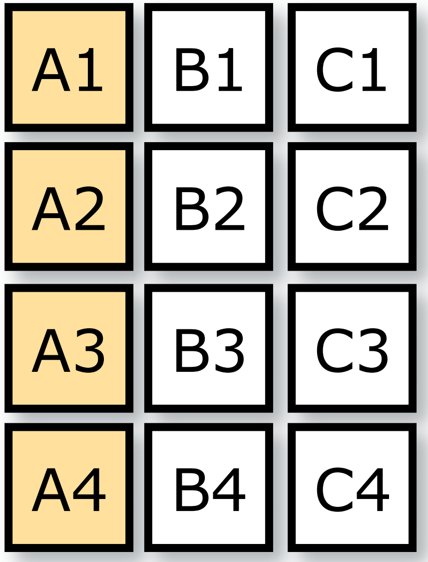
Dictionary Compression

- Variable length fields to fixed length via dictionary compression
- Strides can be reduced and cache line utilization improved



Example: OLAP-Style Query

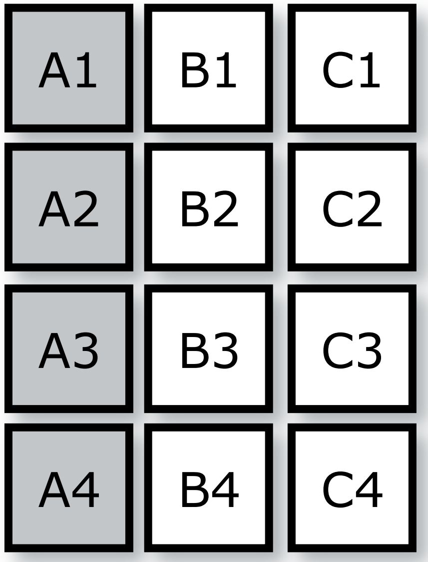
```
struct Tuple {
int a,b,c;
};

Tuple data[4];
fill(data);

int sum = 0;

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

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



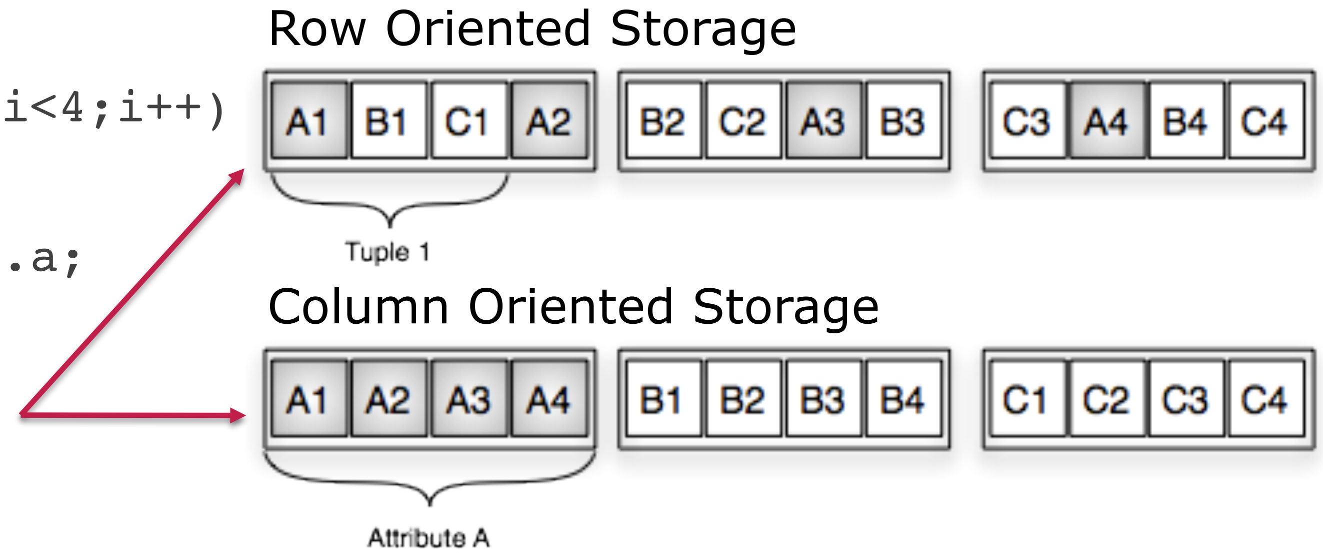
A1	B1	C1
A2	B2	C2
A3	B3	C3
A4	B4	C4

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;
```



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



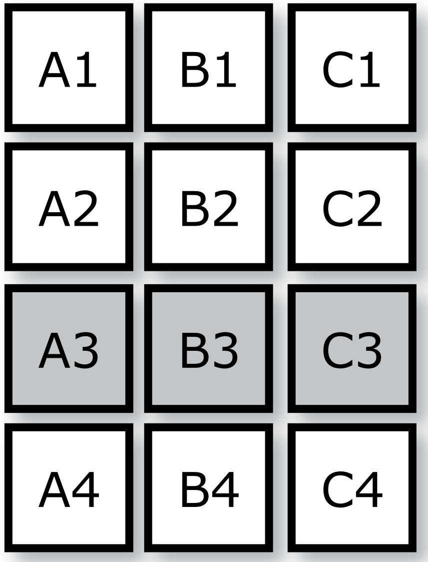
Row Oriented Storage

Column Oriented Storage

Cache line

Example: OLTP-Style Query

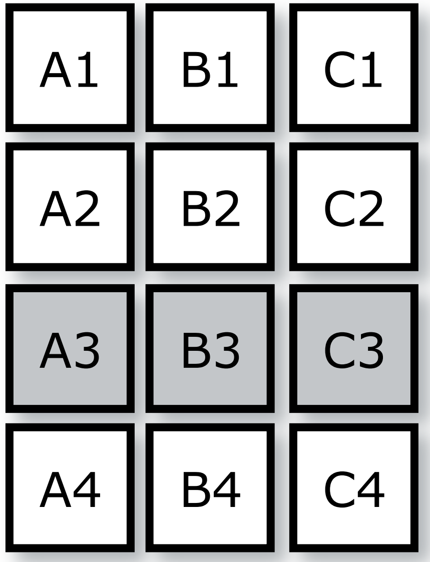
```
struct Tuple {
int a,b,c;
};

Tuple data[4];
fill(data);

Tuple third = data[3];
```



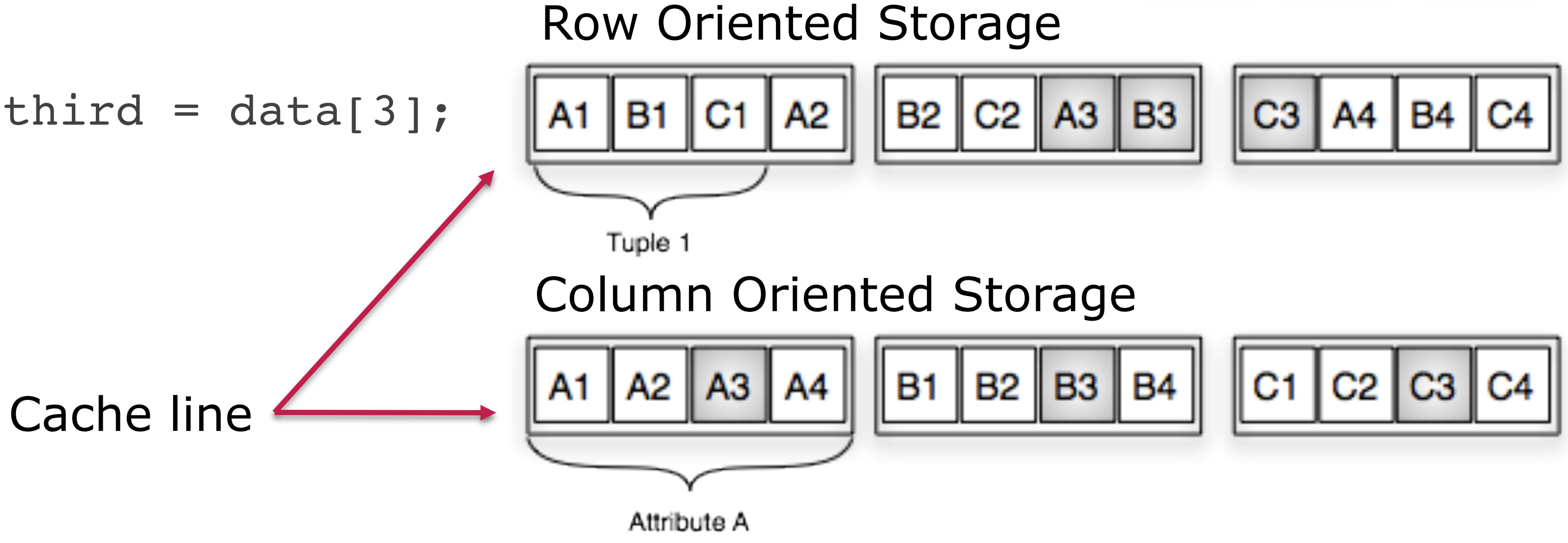
A1	B1	C1
A2	B2	C2
A3	B3	C3
A4	B4	C4

Example: OLTP-Style Query

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

Tuple data[4];
fill(data);

Tuple third = data[3];
```



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



Row Oriented Storage

Column Oriented Storage

Cache line

Questions?