Hasso Plattner Institut

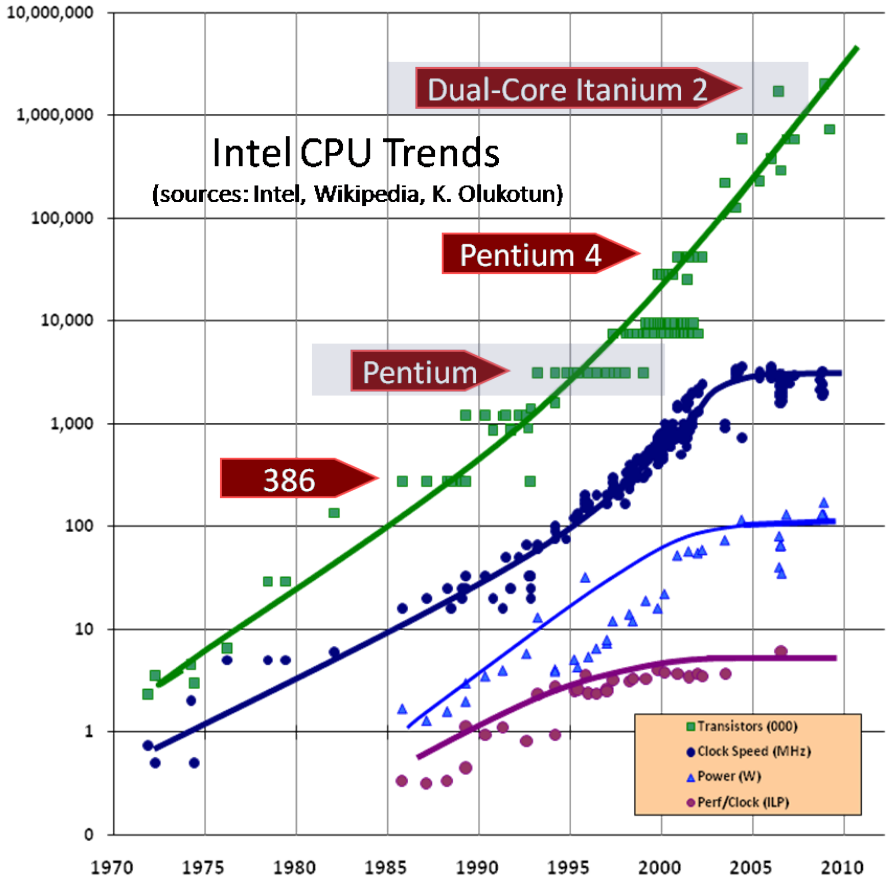
IT Systems Engineering | Universität Potsdam

In-Memory Databases

Algorithms and Data Structures on Modern Hardware

Martin Faust
David Schwalb
Jens Krüger
Jürgen Müller

“The Free Lunch Is Over”



http://www.gotw.ca/publications/concurrency-ddj.htm

- Number of transistors per CPU increases
- Clock frequency stalls

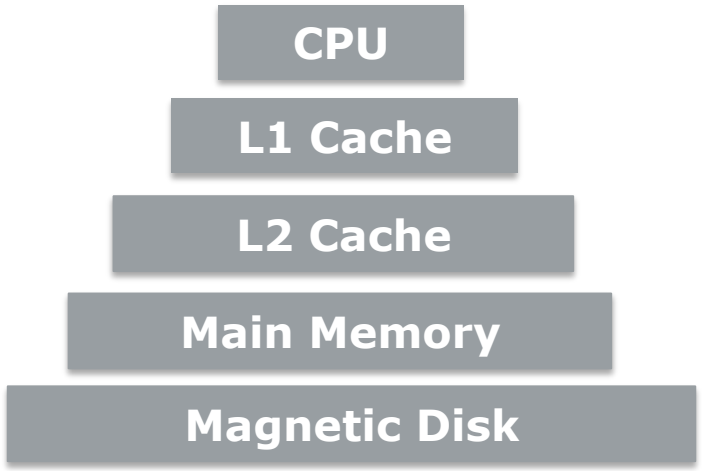
Capacity vs. Speed (latency)

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**

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

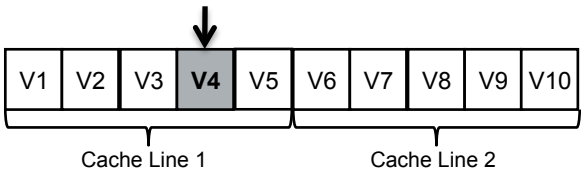
- Not CPU bound
- **But** bandwidth bound
- "I/O Bottleneck"

⇨ 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

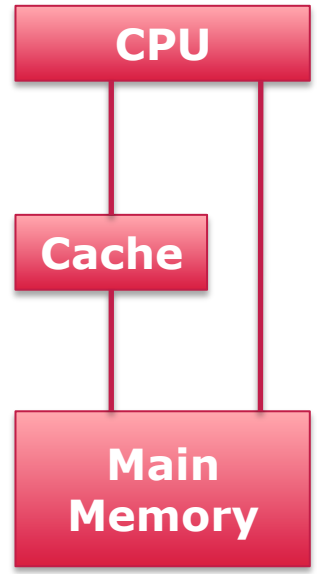
⇨ Potential **waste** of bandwidth



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

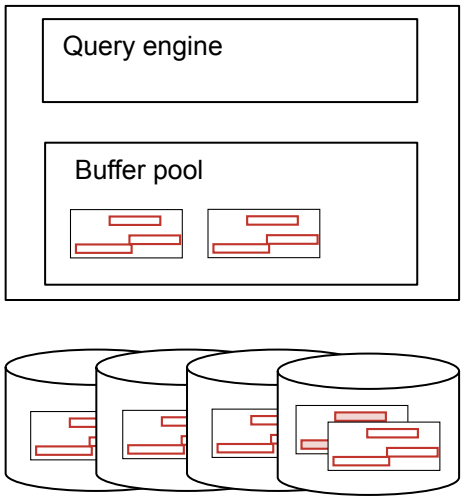
- Non random access (e.g. scan, index traversal):
  - Leverage sequential access patterns
  - Clustering data to a cache lines
  - Partition to avoid cache line pollution (e.g. vertical decomposition)
  - Squeeze more operations/information into a cache line
- Random access (hash join):
  - Partition to fit in cache (cache-sized hash tables)

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



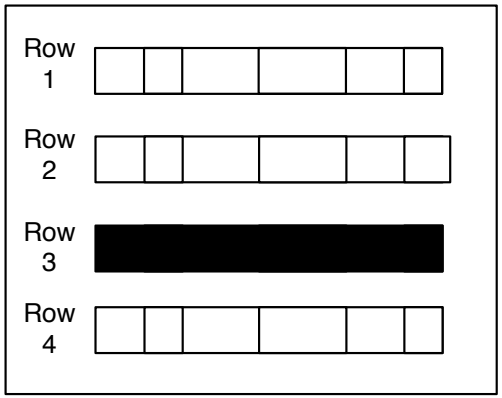
Traditional DBMS Architecture

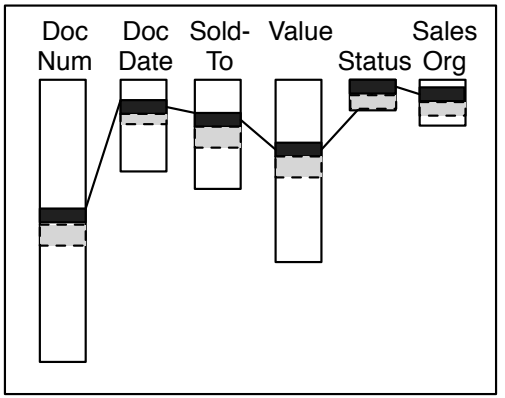
Row- or Column-oriented Storage

Row Store

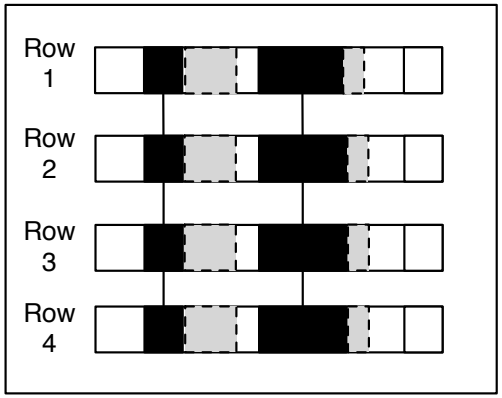
Column Store

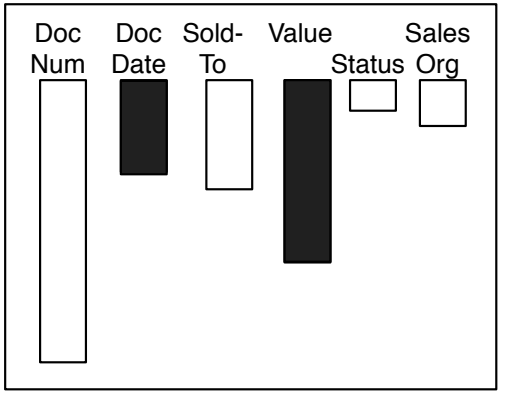
```
SELECT *
FROM Sales Orders
WHERE Document Number = ‘95779216’
```






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





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

Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Seminar Organization

Objective of the Seminar

- Work on advanced database topics in the context of in-memory databases (IMDB) with regards to enterprise data management
  - Get to know characteristics of IMDBs
  - Understand the value of IMDBs for enterprise computing
- Learn how to work scientifically
  - Fully understand your topic and define the objectives of your work
  - Propose a contribution in the area of your topic
  - Quantitatively demonstrate the superiority of your solution
  - Compare your work to existing related work
  - Write down your contribution so that others can understand and reproduce your results

Seminar schedule

- Today: Overview of topics, general introduction
- Tomorrow (17.4.): Q&A on topics, in depth topics
- 22.4.: Send your priorities for topics to lecturers (david.schwalb@hpi.uni-potsdam.de)
- 26.4.: Assignment of topics to students
- 28. & 29.5.: Mid-term Presentations
- 9. & 10.7.: Final Presentations
- 19.7.: Deadline for final documentation

- Throughout the seminar: individual coaching by teaching staff
- Meetings (Room V-1.16)

Final Presentation

- Why a final presentation?
  - Show your ideas and their relevance to others
  - Explain your starting point and how you evolved your idea / implementation
  - Present your implementation, explain your implementations properties

Final Documentation

- 10-12 pages, ACM format [1]
- Suggested Content: Abstract, Introduction into the topic, Related work, Implementation, Experiment/Results, Interpretation, Future Work
- Important!
  - Related work needs to be cited
  - Quantify your ideas / solutions with measurements
  - All experiments need to be reproducible (code, input data) and the raw data to the experiment results must be provided

[1] http://www.acm.org/sigs/publications/proceedings-templates

Grading

- 6 ECTS
- Grading:
  - 30% Presentations (Mid-term 10% / Final 20%)
  - 30% Results
  - 30% written documentation
  - 10% general participation in the seminar

Topic Assignment

- Each participant sends list of top 3 topics in order of preference to lecturers by 22.4.
- Topics are assigned based on preferences and skills by 26.4.

Topics (i) – Data Structures

1) Partitioned Bitvector for In-Memory Column-Store

A bitvector that allows for multiple partitions with different value-id-widths (bits) to create a merge-free insert-only column store. For the seminar, an evaluation of the performance impact for lookups is the main topic.

2) Vertical Bitvector

A modified bitvector storage layout of values on a cacheline to enable faster decompression of multiple values at once due to better utilization of SSE units. [1]

3) Unsorted Dictionaries

Evaluation of a lookup-table scheme, enabling fast range queries with unsorted dictionaries. The idea is to create a cache-sized probing-bitvector by scanning matching positions on the dictionary, which can be accessed fast while scanning the column.

[1] https://github.com/grundprinzip/bitcompressedvector

4) Default Value Index

A data structure to store very sparse columns without sacrificing OLTP performance. The idea is to only store exceptions and a bit-index to mark non-default positions.

5) Multi Column Indices

Classic Multi-Column Indices use a tree-structure to index tuples. In column store this is a cumbersome solution, because values are stored as encoded integers and these are used as long as possible (late materialization), and values that would form one tuple are in different columns.

6) Multi Column Join Algorithms

When joining tables with multiple key columns, column stores need to materialize all keys or need an uncompressed tree structure (actual values not value-ids). The goal is an optimized join algorithm enabling fast joins over multiple columns.

7) Join Algorithms on shared Dictionaries

Columns that store values from the same application domain could be stored with a shared dictionary and joins could be performed directly on value-ids instead of values.

8) Cache-Optimized Parallel Join

Evaluation/Development of a cache-sensitive join algorithm for HYRISE.

9) Analytical User Interface for HYRISE

A graphical, drag and drop based javascript user interface to get fast information about the loaded data, its value distribution, distinct values etc, similar to Visual Intelligence.

10) Set-based Query Language

A query language or graphical query tool that allows a user to define the desired result in a set-based manner, similar to QlikView. The user can define a query by defining subsets of tables etc, natural joins are performed automatically.

Topics (vi)

11) Branching Deltas (Simulation)

For some applications, many of simulations need to be performed. Since they typically require write access, an idea would be to have separate deltas buffers for each simulation. The simulation can then be run in parallel, in order to find the best set of input parameters.

12) Result Set Compression

Currently, result sets are transferred "as is" to the client, where it is simply read row-by-row. In scenarios where the connection between database and applications is limited (mobile, cloud), the idea of compression the result set might lead to significant performance gains.

13) Memory Compression Algorithms in Hardware (PPC)

Modern PowerPC processors have hardware supported main memory compression. A first evaluation could investigate how well it compresses enterprise data, how performance is affected, and how a row store compares to a column store.

14) Rough query runtime prediction based on optimizer results

During development, it is desirable to get rough estimates of how long a query will take to execute. The idea for this seminar is to use the output of the the database query optimizer to roughly estimate the execution time of given SQL queries.

Topics (vii) - Database Co-Processors

15) Execution model for co-processors

Based on the memory bandwidth limitation of database co-processors, multiple execution models are possible. Evaluation of the following different models: Co-Process, Procedural Language Architecture, GPU hosted data.

16) Co-processor integration into databases

Evaluation how a co-processor can be integrated with a database for application specific logic with SAP HANA (AFL) or HYRISE.

17) Automatic execution unit detection in a heterogeneous system

In a heterogeneous system, the capabilities of database (co-)processors differ with respects to bandwidth, parallelization, clock frequency. To determine the best suited execution unit an execution model needs to be created based on device benchmarks and a programs current workflow.

18) Optimized co-processor data structures

Database co-processors used for application specific tasks, such as prediction algorithms or scientific calculations, require the to have a different layout than database tables. The student working on this topic will evaluate which data structures are favorable for different co-processors and algorithms.

19) Automatic code optimization for co-processors

As a heterogeneous programming framework, OpenCL enables the developer to write a single program that can be executed on many different hardware platforms. Nevertheless, the code needs to be optimized for each device to run optimal. The goal will be to optimize the code written in OpenCL C for different devices automatically based on micro benchmarks.