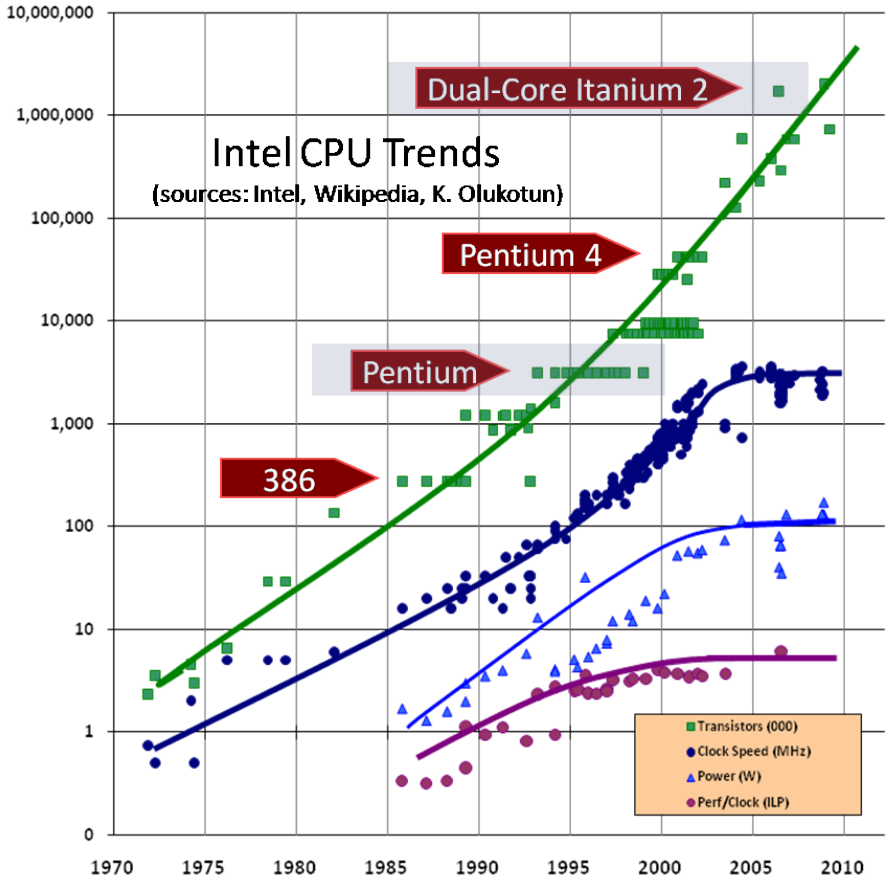
Data Structures and Algorithms for In-Memory Databases

Martin Faust, David Schwalb,
Martin Boissier, Carsten Meyer

“The Free Lunch Is Over”



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

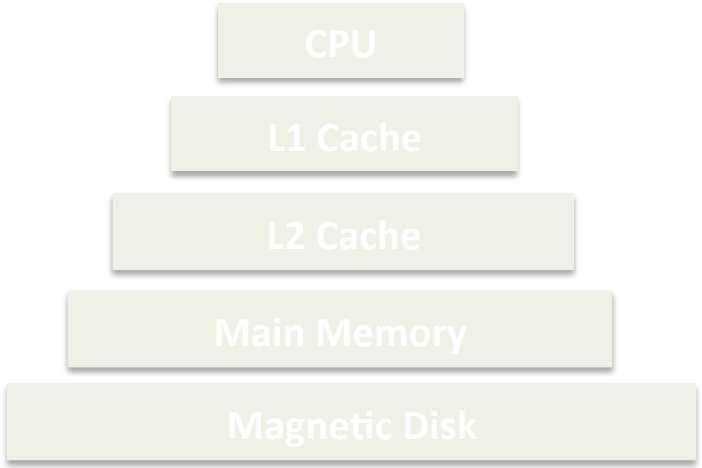
- Number of transistors per CPU increases
- Clock frequency stalls

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**

Data Processing

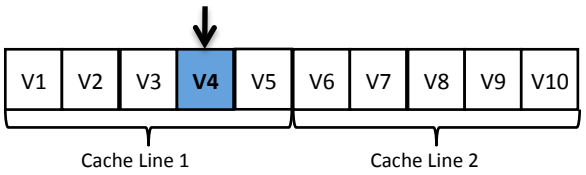
- **In DBMS, on disk as well as in memory, data processing is often:**
  - Not CPU bound
  - **But** bandwidth bound
  - "I/O Bottleneck"

⇒ 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

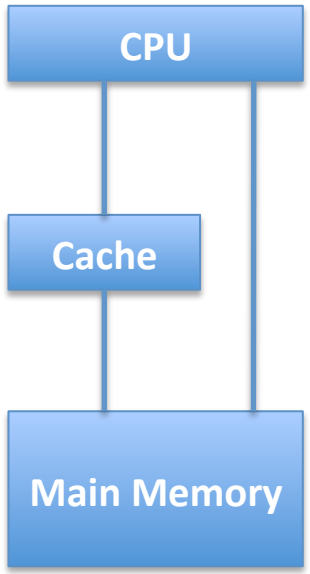
- **Cache**

  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!

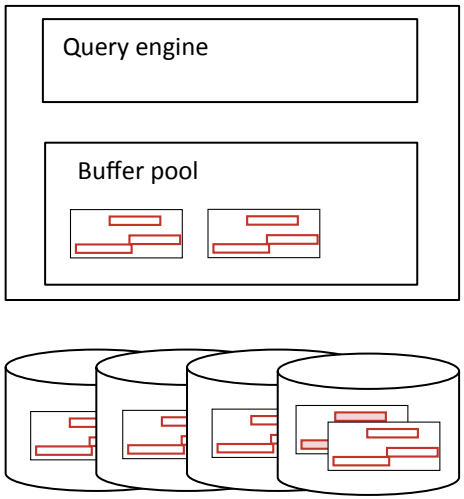
- To improve cache behavior
  - Increase cache capacity
  - Exploit locality
    - Spatial: related data is close (nearby references are likely)
    - Temporal: Re-use of data (repeat reference is likely)
- To improve locality
  - Non random access (e.g. scan, index traversal):
    - Leverage sequential 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)

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



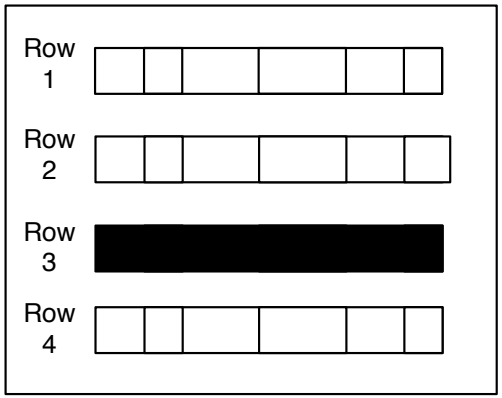
Traditional DBMS Architecture

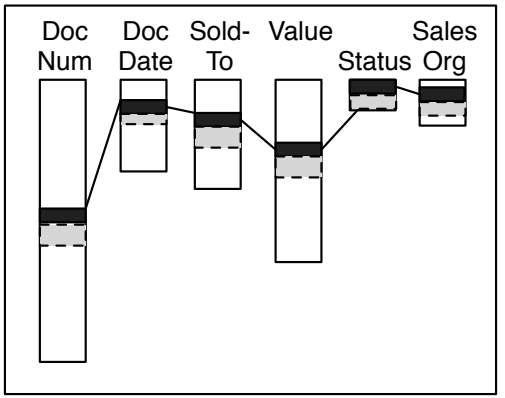
Row- or Column-oriented Storage

Row Store

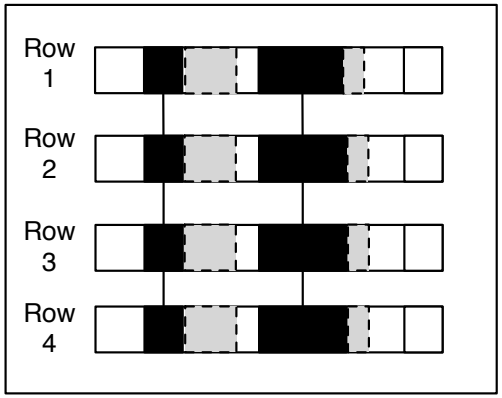
Column Store

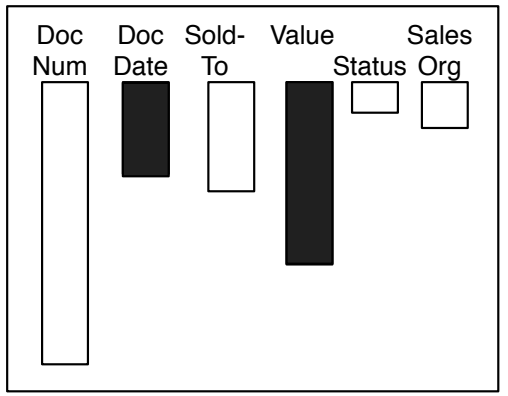
SELECT *
FROM Sales Orders
WHERE Document Number = ‘95779216’





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





Question + Answer

- 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

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 (8.4): Overview of topics, general introduction
- Thursday (10.4.): In-memory DB Basics and Topics Q&A

- 15.4.: Send your priorities for topics to lecturers (martin.faust@hpi.uni-potsdam.de)

- Planned Schedule
  - 20.05.2014: Intro Scientific Writing (tbd)
  - 22./27.05.2014: Mid-term presentation (tbd)
  - 8./10.07.2014: Final presentation (tbd)
  - 26.07.2014: Paper hand-in (tbd)

- Throughout the seminar: individual coaching by teaching staff
- Meetings (Room V-2.16)

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

- 6-8 pages, IEEE 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.ieee.org/conferences_events/conferences/publishing/templates.html

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 15.4.
- Topics are assigned based on preferences and skills by 24.4.

HYRISE

- Open source IMDB
- Hosted at github.com/hyrise
- C++ 11
- Query Interface: Query plan or stored procedures

Recommended Papers for Intro

- Plattner, SIGMOD 2009: *A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database*
- Grund et al. VLDB 2010: *HYRISE—A Main Memory Hybrid Storage Engine*
- Krueger et al. VLDB 2012: *Fast Updates on Read-Optimized Databases Using Multi-Core CPUs*

Topics

Primary Key / Unique Index

- Currently PK/FK/Unique constraints are not enforced in HYRISE
- Goal is to research, evaluate, implement and measure different approaches
- Possible implementation tasks:
  - Resorting of the table by its key (clustered index)
  - Probabilistic data structures
  - Additional hash, tree or trie structures

Trie Index / Radix Tree

- A radix tree is a space efficient index structure
- Build & integrate into Hyrise and compare the storage space and performance to alternatives.
- To achieve maximum performance the student should implement a lock free radix tree structure, using for example HLE or transactional memory

Compound Indices

- Improve current merge implementation to support the efficient merge of multi-column indices.
- Determine performance, both theoretically and through benchmarks
- Quantify parameters that impact the merge
- Tune index implementation

Leveraging FusionIO Auto Commit Memory as Persitency of In-Memory Databases

- Auto Commit Memory (ACM) allows to map storage directly into memory, to modify it on a byte level and to automatically sync changes back to storage
- Task: Integrate ACM into Hyrise, measure performance impact
  - Working with newest FusionIO drive
  - Building on existing NVM integration in Hyrise

Memory Mapped File Checkpointing

- A checkpoint is a consistent snapshot of the database to speed up recovery
- In-memory databases with main/delta concept need to write complete delta to storage for checkpoint
- Task: Implement checkpoint algorithm in Hyrise, by allocating delta data structures on Memory Mapped files on a Fusion ioDrive and perform a *msync()* of the file for the checkpoint
  - Working with newest FusionIO drive
  - Measure performance implications
  - Compare with 'normal' serialization of delta to storage

Optimized Logging for Flash

- Traditional ARIES style logging with group commits is optimized for shortcomings of disks
- Modern flash based storage devices have completely different characteristics
- Task:
  - Optimize the logging algorithm in Hyrise for flash storage drives
  - Analyse the performance of different implementations

Database Cockpit for Hyrise

- Task:
  - Develop an HTML5 application that visualizes heartbeats from Hyrise during the execution of a TPC-C workload
  - Showing multiple live charts visualizing query performance and database statistics
  - Control elements to influence workload are provided by the app, as well as the ability to simulate a crash in the database and show instant recovery features

Benchmark Framework

- Currently available Python-based benchmark framework allows the execution of a TPC-C workload on Hyrise
- Benchmark tool using the Apache Benchmark Tool (ab) to send queries to database
- Task:
  - Improve benchmark tool to support multiple workloads and modular experiments
  - e.g. CH-Benchmark,TPC-CH, TATP, TPC-C

Multiple Event-Loops for Hyrise

- Current Hyrise server is using libev and libebb as basic event loop to dispatch incoming queries
- Current implementation runs one event loop, however dispatching short running queries onto different NUMA nodes is expensive
- Task:
  - Implement multiple event loops for Hyrise listening all on one private HTTP port with central dispatch mechanism to redirect incoming connections to the respective ev-loops
  - Alternatively, instead of using HTTP, a custom protocol might be developed

Shared Domain Dictionary for HYRISE

Improved Join and Update Performance

- Order-preserving dictionaries (e.g. HYRISE, SAP HANA)
  - inefficient mapping structures for cross table operations (e.g. JOIN)
  - costly data re-encoding during merge
- Findings
  - Join operations always between columns of the same domain
  - Value-ranges of PK columns are typically incremental (but not FK)
- Idea
  - A shared dictionary (encoding) for PK and FK of the same domain
  - Direct join on (compressed) valueIDs
  - No re-encoding during merge for PK/ FK columns.
- *Task*
  - *Implement a shared domain dictionary (SDD) as well as an adapted merge and join operation*
  - *Evaluate performance using HYRISE*

Tiering-Indices in HYRISE

Analyzing SQL-Traces for Hot and Cold Data Access

- HYRISE tables can be partitioned horizontally & vertically in order to:
  - use different memory tiers (Storage Class Memory)
  - limit operations to certain data areas
- *Extract characteristics from DB workloads and build index structures that:*
  - classify hot data
  - identify hot-only queries
- Setting:
  - Workloads and data from openHPI and an enterprise system
  - Work with current bachelor-project team
  - Statistics and data mining (co-entropy, distribution, time-related patterns, etc.)

Thank you.