FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Datenbanksystemimplementierung

Prof. Dr. Viktor Leis

WS 2019/2020

Professur für Datenbanken und Informationssysteme

Introduction

Introduction

Database Management Systems (DBMS) are extremely important

- used in nearly all commercial data management
- very large data sets
- valuable data

Key challenges:

- scalability to huge data sets
- reliability
- concurrency

Results in very complex software.

About This Lecture

Goals of this lecture

- learning how to build a modern DBMS
- understanding the internals of existing DBMSs
- understanding the effects of hardware behavior
- learn how to write efficient and scalable software

Literature

- Theo Härder, Erhard Rahm: *Datenbanksysteme - Konzepte und Techniken der Implementierung*. Springer-Verlag, 2001.
- Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom: *Database Systems: The Complete Book*. Prentice-Hall, 2008.

Unfortunately mainly cover the classical architecture.

Motivational Example

Why is a DBMS different from most other programs?

- many difficult requirements (reliability etc.)
- but a key challenge is **scalability**

Motivational example

Given two lists L_1 and L_2, find all entries that occur on both lists.

Looks simple...

Motivational Example (2)

Given two lists L_1 and L_2, find all entries that occur on both lists.

Simple if both fit in main memory

Motivational Example (2)

Given two lists L_1 and L_2, find all entries that occur on both lists.

Simple if both fit in main memory

- sort both lists and intersect
- or put one in a hash table and probe
- or build index structures
- or ...

Note: pairwise comparison is not an option! $O(n^2)$

Motivational Example (3)

Given two lists L_1 and L_2, find all entries that occur on both lists.

Slightly more complex if only one fits in main memory

Motivational Example (3)

Given two lists L_1 and L_2, find all entries that occur on both lists.

Slightly more complex if only one fits in main memory

- load the smaller list into memory
- build index structure/sort/hash/...
- scan the larger list
- search for matches in main memory

Code still similar to the pure main-memory case.

Motivational Example (4)

Given two lists L_1 and L_2, find all entries that occur on both lists.

Difficult if neither list fits into main memory

Motivational Example (4)

Given two lists L_1 and L_2, find all entries that occur on both lists.

Difficult if neither list fits into main memory

- no direct interaction possible
- sorting works, but already a difficult problem
- or use some kind of partitioning scheme
- breaks the problem into smaller problem
- until main memory size is reached

Code significantly more involved.

Motivational Example (5)

Given two lists L_1 and L_2, find all entries that occur on both lists.

Hard if we make no assumptions about L_1 and L_2.

Motivational Example (5)

Given two lists L_1 and L_2, find all entries that occur on both lists.

Hard if we make no assumptions about L_1 and L_2.

- tons of corner cases
- a list can contain duplicates
- a single duplicate might exceed main memory!
- breaks “simple” external memory logic
- multiple ways to solve this, but all somewhat involved
- and a DBMS must not make assumptions about its data!

Code complexity is very high.

Motivational Example (6)

Designing scalable algorithm is a hard problem

- must cope with very large instances
- hard even in main memory
- billions of data items
- rules out any $O(n^2)$ algorithm
- external algorithms are even harder

This requires a careful software engineering.

Set-Oriented Processing

Small applications often loop over their data

- one for loop accesses all item x,
- for each item, another loop access item y,
- then both items are combined.

This kind of code of code feels “natural”, but is bad

- $\Omega(n^2)$ runtime
- does not scale

Instead: **set oriented** processing. Perform operations for large batches of data.

Set-Oriented Processing (2)

Processing whole batches of tuples is more efficient:

- can prepare index structures
- or re-organize the data
- sorting/hashing
- runtime ideally $O(nlogn)$

Many different algorithms, we will look at them later.

Traditional Assumptions

Historically, DBMS are designed for the following scenario:

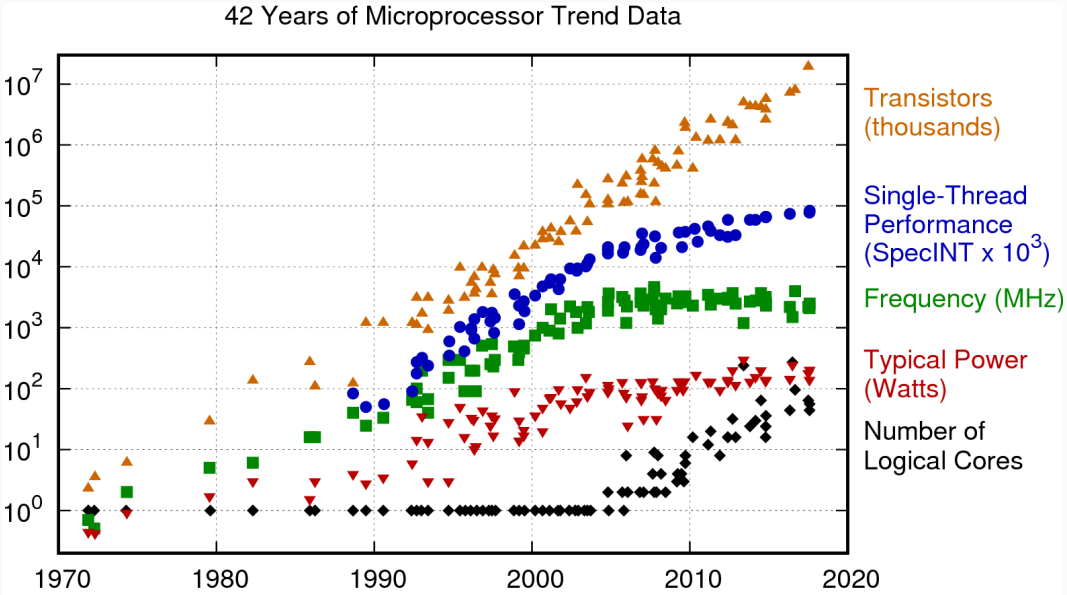
- data is much larger than main memory
- I/O costs dominate everything
- random I/O is very expensive

This led to a very **conservative**, but also very **scalable** design.

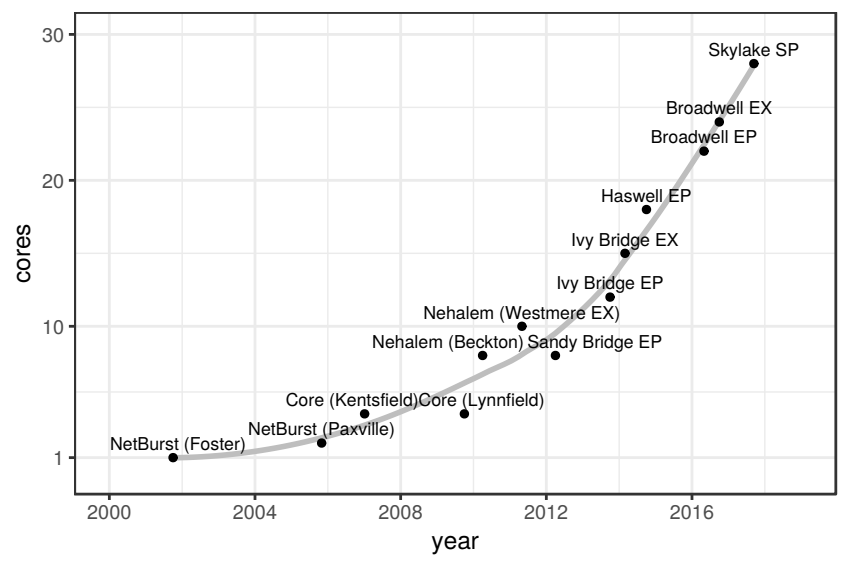
Hardware Trends

- in the past, single-threaded performance doubled every 18-22 months
- now single-threaded performance is stagnating (single digit percentage growth)
- number of transistors is still growing (“Moore’s law”)
- as a result, chips offer more and more parallelism
- to benefit from this parallelism, software must generally be rewritten (“the free lunch is over”)
- software is becoming “a producer of performance” instead of a “consumer of performance” (Mark Hill)

Hardware Trends

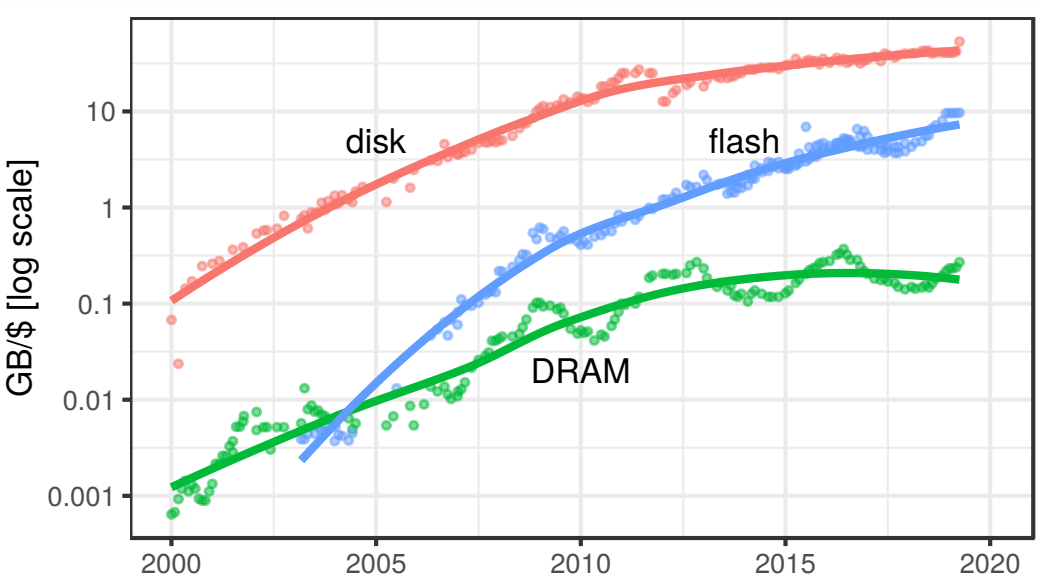


Number of Cores



Single Instruction, Multiple Data (SIMD) Width

- 1997: MMX 64-bit (Pentium 1)
- 1999: SSE 128-bit (Pentium 3)
- 2011: AVX 256-bit float (Sandy Bridge)
- 2013: AVX2 256-bit int (Haswell)
- 2017: AVX-512 512 bit (Skylake Server)


GB/$ [log scale]
10
1
0.1
0.01
0.001
2000
2005
2010
2015
2020
disk
flash
DRAM

What do $2000 buy in 2019?

	config	capacity [TB]	bandwidth read [GB/s]	bandwidth write [GB/s]	random reads [M/s]
disk	20×4 TB	72.8	2	2	0.002
flash	8×1 TB	7.3	25	20	4
persistent mem	3×128 GB	0.4	37	10	155
main memory	12×32 GB	0.4	92	80	1,500

Hardware Trends

Hardware development changed some of the assumptions

- main memory size is increasing
- servers with 1TB main memory are affordable
- flash storage reduces random I/O costs
- ...

This has consequences for DBMS design

- CPU costs become more important
- often I/O is eliminated or greatly reduced
- the old architecture becomes suboptimal

But this is more evolution than revolution. Many of the old techniques are still required for scalability reasons.

Goals

Ideally, a DBMS

- handles arbitrarily large data sets efficiently
- never loses data
- offers a high-level API to manipulate and retrieve data
- shields the application from the complexity of data management
- offers excellent performance for all kinds of queries and all kinds of data

This is a very ambitious goal!
In many cases indeed reached, but implies complexity.

Overview (Preliminary)

1. The Classical Architecture
   - 1.1 storage
   - 1.2 access paths
   - 1.3 transactions and recovery
2. Efficient Query Processing
   - 2.1 set oriented query processing
   - 2.2 algebraic operators
   - 2.3 code generation
3. Designing a DBMS for Modern Hardware
   - 3.1 re-designing storage
   - 3.2 optimizing cache locality
   - 3.3 main memory databases

Programming Assignments

- this course is about implementation skills, not just about abstract knowledge
- there will be C/C++ assignments
- the assignments are not mandatory
- but most of the learning happens through them
- this course can change your career