Cloud-Based Data Processing

Introduction

Jana Giceva

About me

Jana Giceva
Chair for Database Systems
Boltzmannstr. 3, Office: 02.11.043
jana.giceva@in.tum.de

Academic Background:

- 2011 – 2017 PhD in Computer Science at *ETH Zurich* (topic: DB/OS co-design)
- 2017 – 2019 Lecturer in Department of Computing at *Imperial College London*
- Since 2020 Assistant Professor for Database Systems at *TUM*

Connections with Industry:

- Held roles with *Oracle Labs* and *Microsoft Research* in the USA in 2013 and 2014
- PhD Fellowship from *Google* in 2014
- Early Career Faculty Award from *VMware* in 2019

What this course is about

- ***Learn* how to design scalable and efficient cloud-native systems**
  - Understand the demands of **novel** (data) **workloads** and the **economies** and **challenges** at **scale**
  - Get to know the **internals** of modern **data centers** and emerging technologies and trends
  - Learn the **fundamental principles** for building **scalable system** software

- ***Build* a cloud-native multi-tier data processing system:**
  - Work across multiple layers of the stack: storage, synchronization, caching, compute, etc.
  - Tailor the system for given workload requirements
  - **Think** in terms **of performance**, **scalability**, **fault tolerance**, **elasticity**, **high availability**, **cost**, **privacy**, etc.
  - Use modern cloud constructs like containers or serverless functions.

- ***Apply* the knowledge with hands-on work:**
  - Modular homework assignments
  - Individual project work

Motivation

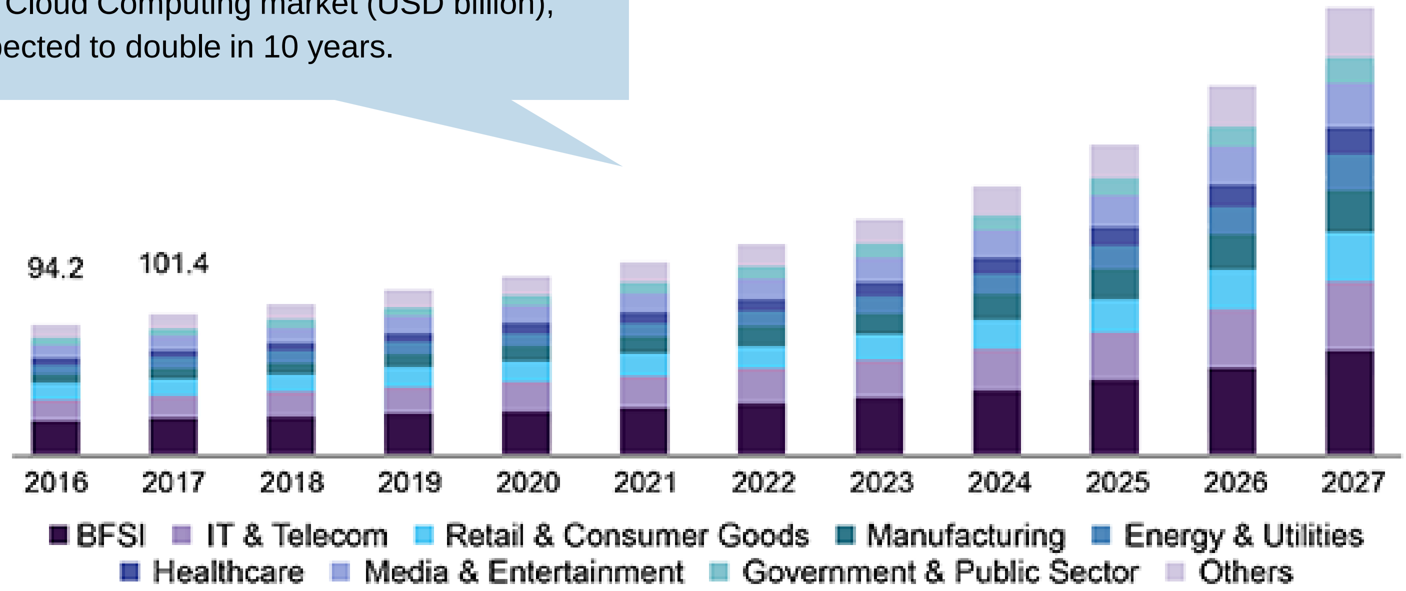
Motivation

- Why should we care about the cloud?
- What impact does the cloud have on system development?
- Why should we focus on data-processing in particular?

Why is Cloud important?

- The internet has around **4.5 billion users** today, and the number is still growing
- **Digitalization** of society and the **Cloud transform** whole **industries**
- **25%** increase in cloud usage during the pandemic (src: Gartner 2022)

US Cloud Computing market (USD billion), expected to double in 10 years.



https://www.grandviewresearch.com/industry-analysis/cloud-computing-industry

https://99firms.com/blog/google-search-statistics

How the Cloud impacts technology development?

- Cloud helps in fast dissemination of new technologies
- Easy, fast and cheap exposure to new trends available for everyone

- **Accelerators**

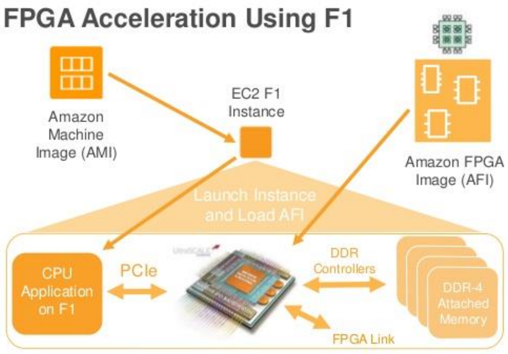
  EC2 offers instances with the latest GPUs, custom ML inference ASICs or FPGA, TPUs

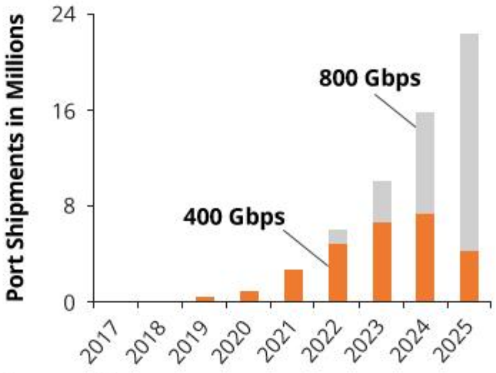
Fast network interconnects

c6gn.16xlarge already offers 64 cores, 128 GiB memory and 100Gbps network for $2.8 per hour

Latest storage technologies

Microsoft's revolutionary glass storage with [Project Silica]() or Holographic storage (HSD)





*source: Dell'Oro Group

Cloud providers control the full stack

- **Influence the hardware landscape**
  - Innovation from novel chip design, to new switches and network fabrics, incl. storage technologies
- **Control the full software stack**
  - they can change or customize it (OS, virtualization, containers, etc.)
- **Introduce or popularize new programming methodologies and paradigms**
  - Map-Reduce, actor-based programming models, microservices and serverless, etc.
- **Revolutionize how we approach application design and implementation**
  - Scale, elasticity, cost, privacy, etc.

How are things different at scale?

As reported by Google (slides from Jeff Dean) in 2010:

Focus is more on meeting the SLOs (service-level objectives) with respect to:

- Performance (latency)
- High availability
- Efficiency
- Elasticity

Most complexity is absorbed by the cloud system software infrastructure

The Joys of Real Hardware

Typical first year for a new cluster:

~1 network rewiring (rolling ~5% of machines down over 2-day span)

~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)

~5 racks go wonky (40-80 machines see 50% packetloss)

~8 network maintenances (4 might cause ~30-minute random connectivity losses)

~12 router reloads (takes out DNS and external vips for a couple minutes)

~3 router failures (have to immediately pull traffic for an hour)

~dozens of minor 30-second blips for dns

~1000 individual machine failures

~thousands of hard drive failures

slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

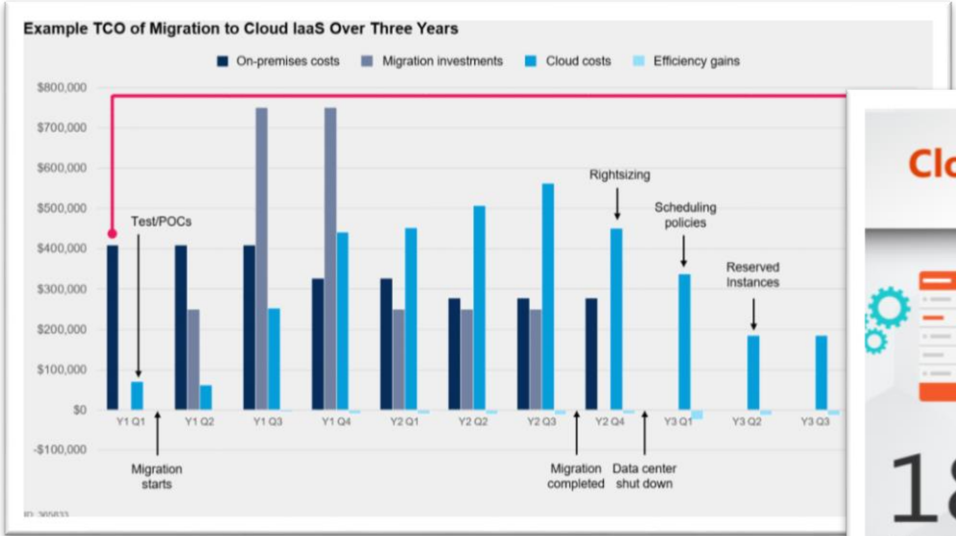
Reliability/availability must come from software!

Google

https://static.googleusercontent.com/media/research.google.com/en//people/jeff/Stanford-DL-Nov-2010.pdf

But it is not just scale!

- Incentives highly driven by reduction of cost
- Skeptics primarily worried about cloud's privacy and security.



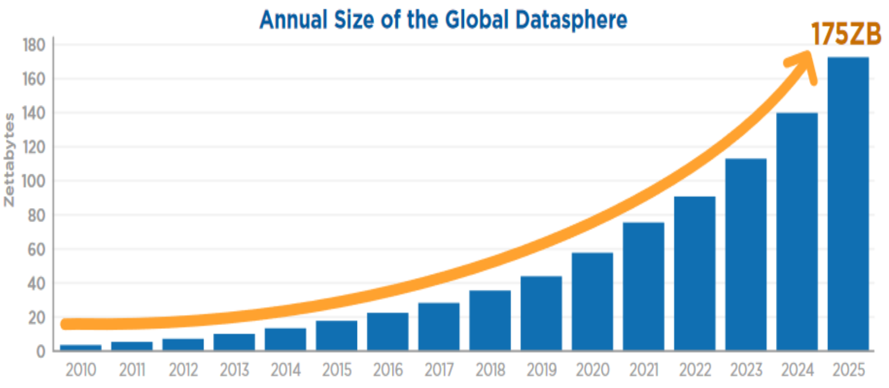
https://blogs.gartner.com/marco-meinardi/2018/11/30/public-cloud-cheaper-than-running-your-data-center/

https://dzone.com/articles/data-security-an-integral-aspect-of-cloud-computin

Why focus on data-processing?

- Surge in data volumes produced and consumed

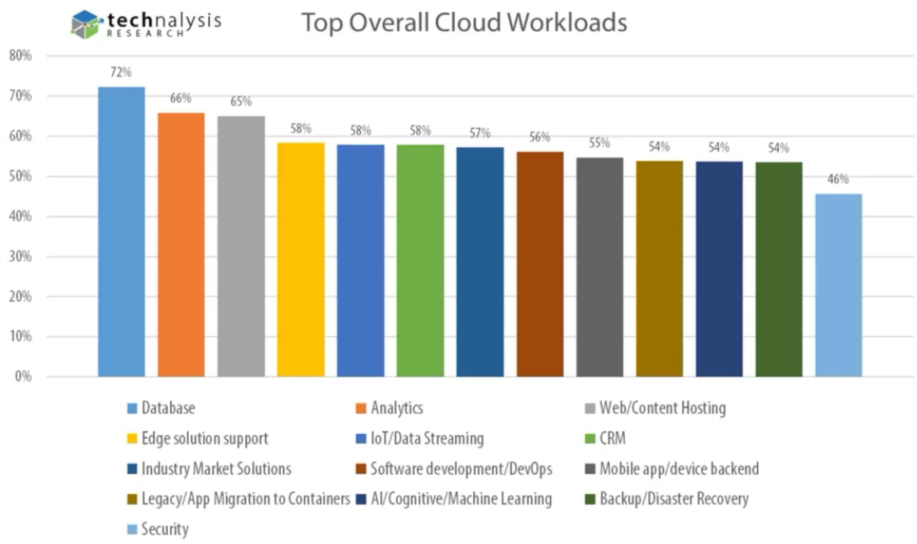
Figure 1 - Annual Size of the Global Datasphere



Source: Data Age 2025, sponsored by Seagate with data from IDC Global DataSphere, May 2020

https://www.seagate.com/files/www-content/our-story/trends/files/dataage-idc-report-final.pdf

- Data-processing still the dominant workload:
  - Databases, analytics, streaming, etc.



https://www.techspot.com/news/83646-companies-taking-advantage-different-cloud-options-putting-different.html

Course administrivia

Course content

- **Data centers and cloud computing**
- **Distributed data basics (partitioning, replication, fault-tolerance, consistency, consensus)**
- **Design principles for cloud-based applications**
- **Design and build scalable systems for the cloud:**
  - Covering **storage, query, and transaction processing.**
- **Trends, emerging technologies and their impact on the future of cloud-systems**
  - Hardware and accelerators, resource disaggregation, software-defined networking/storage

Special focus on **state-of-the-art systems** that are **used in production**

Course Organization

Lecture:

- **In-person** lectures on **Thursdays 2-4pm (Galileo 8120.EG.001)**
  - **Slides** uploaded on course web-page and moodle (by Thursday noon).
  - Old lecture video recordings from WS 20/21 available on moodle.
- Course **website**: https://db.in.tum.de/teaching/ws2324/clouddataprocessing/
- Please check regularly for updates

Tutorials:

- **In-person** tutorials after the lectures
- **Thursdays 4-5pm (Galileo 8120.EG.001)** – not recorded
- **TAs** for the course are
  Michalis Georgoulakis (michalis.georgoulakis@tum.de)
  Tobias Götz (goetzt@in.tum.de)
- First session: today for introduction, Q&A session and general set-up
- Consider that **exercise material** is **part** of the course **content**!

Assignments and Project

- The main goal of the course is **critical thinking** and analyzing the main **design decisions** behind **scalable systems** and understanding what it takes to build them.

- **The assignments will give you a range of different skillsets:**
  1. Analysis on different design decisions on how to build a data processing system in the cloud
  2. Measurement study on existing cloud services, system design and back-of-the-envelope calc.
  3. Hands-on implementation of a data processing task that uses the cloud services you benchmarked.

- You can then **apply them** for your **project** in the last 5 weeks of the course.

Assessment and Exam

- Bonus: extra points for the 90min final exam

- Maximum bonus: 14 points
  - Homework assignments: up to 7 points
  - Project: up to 7 points

- Passing criteria:
  - Exam needs to be passed so the bonus points can be accounted for
  - For the homework assignments – details later in the tutorial session

- Written exam (details to be announced later)

Course Set-up

Let's make the course as interactive as possible

- During the lecture and tutorials, please speak-up, ask questions and discuss!
- Also engage in asynchronous discussions on Mattermost
- Send the TAs questions you want to be addressed during the tutorial sessions

The material we discuss is relevant in practice:

- We will provide examples
- You will achieve the maximum fun factor if you do the project work

- We will have a **few guest speakers** (also from **industry**)
  - Details to be announced later in class.

Course material

This is not a standard course – it is state of the art (bleeding edge) systems and research

- There is **no real textbook** for this course, but a good overview of the principles behind **building scalable systems** are covered in:
  - "Designing Data-Intensive Applications" by Martin Kleppmann
  - "Azure Application Architecture Guide" by Microsoft
  - "Architecting for the Cloud" by AWS
- More on **hardware-** and **software-virtualization** is covered in:
  - "Hardware and Software Support for Virtualization" by Ed Bougnon, Jason Nieh, and Dan Tsafrir.

- The **lecture slides** are available **online**
- Most **material** that we are going to cover is **taken out of research papers:**
  - The references to those papers (all good, easy and fun! to read) will be given as we go.
  - Relevant conferences: ACM/USENIX SOSP/OSDI, ACM SOCC, USENIX ATC, NSDI, ACM EuroSys, ACM SIGMOD, VLDB, ACM SIGCOMM, IEEE ICDE, ACM CoNEXT, etc.

Cloud-based application design

Challenges

Distributed Computing Challenges

Scalability

- Independent parallel processing of sub-requests or tasks
- E.g., adding more servers permits serving more concurrent requests

Fault Tolerance

- Must mask failures and recover from hardware and software failures
- Must replicate data and service for redundancy

High Availability

- Service must operate 24/7

Consistency

- Data stored / produced by multiple services must lead to consistent results

Performance

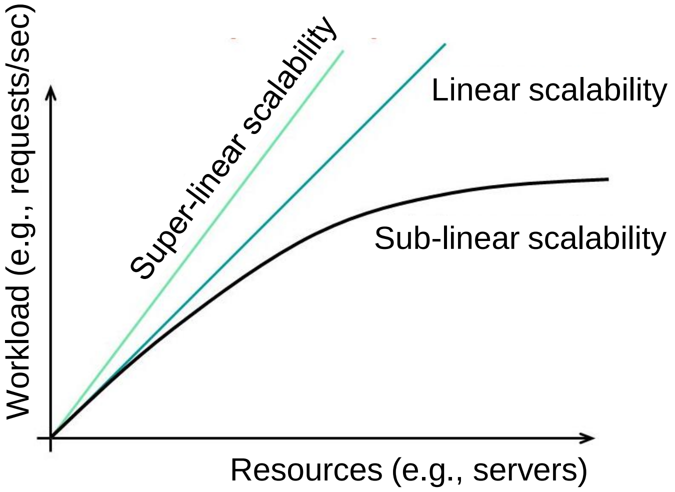
- Predictable low-latency processing with high throughput

Scalability matters

Ideally, adding N more servers should support N more users!

But, **linear scalability** is **hard** to achieve:

- Overheads + synchronization
- Load-imbalances create hot-spots
  (e.g., due to popular content, poor hash function)
- Amdahl's law → a straggler slows everything down



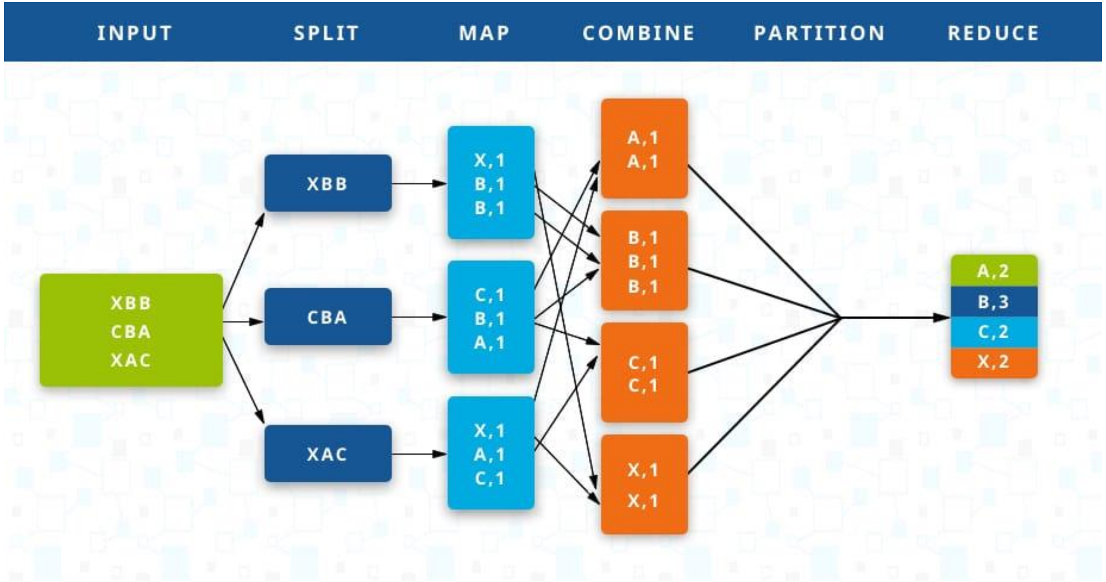
Therefore, one needs to **partition both data** and **compute.**

Scaling computation

How do data-intensive applications scale?

- Enable task-parallel or data-parallel processing
- Frontend does the aggregation of (select top-k documents)
- Back-ends provide partial responses

e.g., Map-Reduce



Img src: https://www.talend.com/de/resources/what-is-mapreduce/

Fault tolerance

- **Think of failure as the common case.**

- **Full redundancy** is too **expensive** → **use failure recovery**.
  - Impossible to build redundant systems at scale
  - Rather reduce the cost of failure recovery

- Failure recovery: **replication** or **re-computation**
  - Which one is better, depends on the respective costs

- **Replication**:
  - Need to replicate data and service
  - Introduces the consistency issues

- **Re-computation**
  - Easy for stateless services
  - Remember data lineage for compute jobs

High availability

- **Downtime** → **bad** customer **experience**, and **loss** in **revenue**.
- According to Gartner, a **minute** of IT **downtime** costs companies **$5’600 on average**.

Cloud service providers offer **service level agreements (SLAs)** to their clients.
A **commitment/contract** for the **quality** of the **service** (e.g., availability, performance, etc.)

Translating downtime for a typical SLA for availability:

- **99.9%** (“three nines”) availability means **8.77 hours downtime per year** → close to $3 million.
- **99.99%** (“four nines”) availability means **52.6 minutes downtime per year** → close to $300’000.

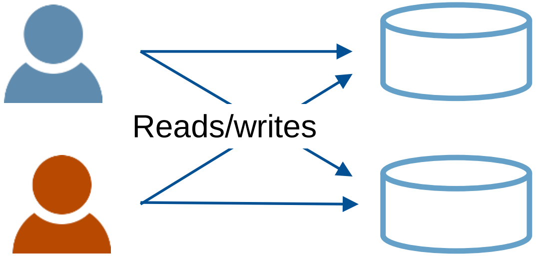
For a **high available** service one needs to design and:

- Eliminate **single point of failure** by adding redundancy in the system.
- Have a **reliable crossover.**
- Have an efficient way to **monitor** and **detect failures** when they occur.

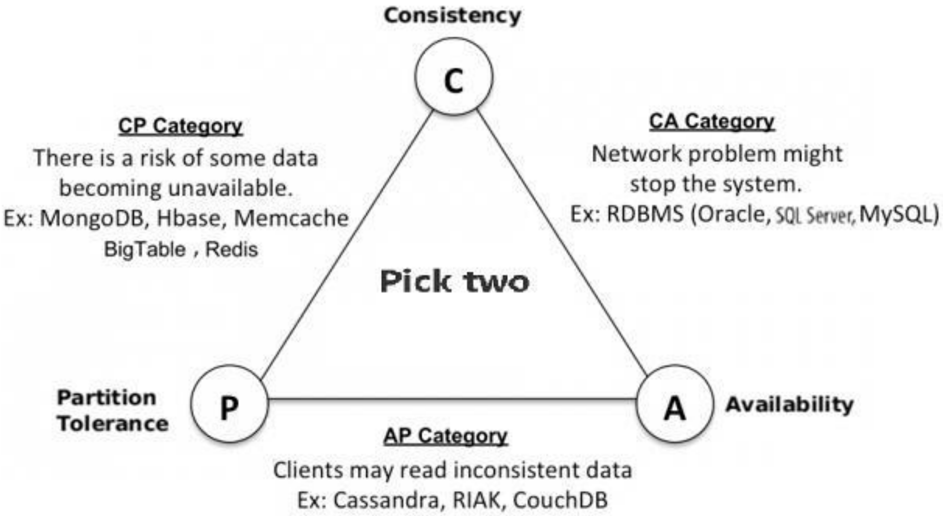
e.g., Amazon S3 offers 11 9s of availability of objects across multiple availability zones (AZs).

Consistency

- Many **applications** need **state replicated** across a wide area, for reliability, availability and low latency.



- **CAP Theorem:** *It is impossible for a distributed data store to simultaneously provide more than two out of the three guarantees*:
  - Consistency
  - Availability
  - Partition tolerance



Src: CAP Theorem by Eric Brewer, formally defined by Gilbert and Lynch

Consistency models

- **Two main choices:**
  - **Strongly consistent** operations (e.g., use Paxos, Raft, etc.)
    - Often at the cost of additional latency for the common case
  - **Inconsistent** operations
    - Better performance / availability, but applications are harder to write and reason about the model

- **Many applications aiming for high availability gravitated towards eventual consistency**
  - E.g., **Gmail**: marking a message as read is asynchronous, but sending a message needs to be a consistent operation
  - Order of posts in **LinkedIn** news feed? Access from multiple devices?
  - Count of song popularity in **Spotify**?

- But, modern data analytics (data lakes, training ML on PBs of data) require strong consistency
  https://www.allthingsdistributed.com/2021/04/s3-strong-consistency.html

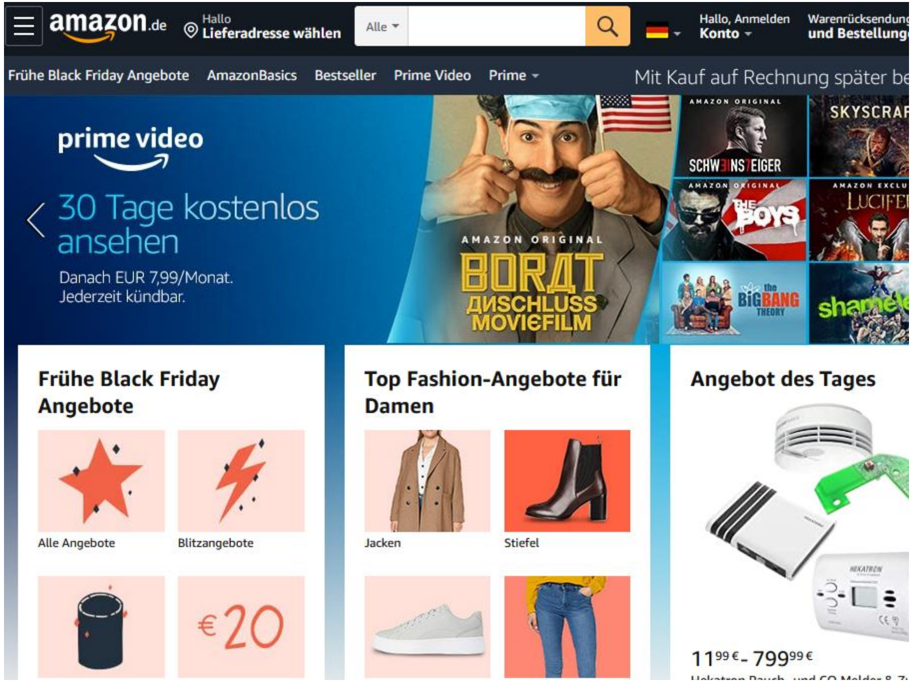
"Eventual Consistency Today: Limitations, Extensions, and Beyond" by Bailis and Ghodsi in ACM Queue vol. 11, issue 3, 2013

Performance matters

Online services (e.g., Facebook, Google search, Bing):

- Expected response time < 100ms

https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/



Performance affects revenue:

- Values reported 10 years ago
  - **Amazon**: every 100ms of latency costs them 1% in sales
  - **Google** found an extra 0.5 secs drops traffic by 20%
- Akamai in 2017 found that a 100ms delay in page load time results in 6% drop in sales
- Even more valid **today** in mobile web browsing/app responsiveness

The tail at scale

- At scale, looking at the average request latency is **not** enough.
- **Tail latency** = the last 0.X% of the request latency distribution graph.
  – e.g., we can take the slowest 1% response times or the 99%ile response time.
- Tail latency is **amplified** by **scale**, due to **fan-outs** for
  – **Micro-services**, **data partitions**
- Overall latency ≥ latency of the **slowest** component
- Servers with 1ms average, but 1sec 99%ile latency
  – 1 server: 1% of the requests take >= 1 sec
  – 100 servers: 63% of the requests take >= 1sec

"The Tail at Scale" by Jeffrey Dean and Luiz Andre Barroso in *Comm. Of the ACM, 2013*
Tail Latency Might Matter More than you Think – Marc Brooker blog.

The tail at scale

- Increased fan-out has a large impact on the latency distributions.
- At Google scale:
  - 10ms 99% percentile for any single request
  - The 99% percentile for all requests is 140ms and the 95% percentile is 70ms
    - Waiting for the slowest 5% of the requests accounts for half of the total 99% percentile latency.

Table 1. Individual-leaf-request finishing times for a large fan-out service tree (measured from root node of the tree).

	50%ile latency	95%ile latency	99%ile latency
One random leaf finishes	1ms	5ms	10ms
95% of all leaf requests finish	12ms	32ms	70ms
100% of all leaf requests finish	40ms	87ms	140ms

"The Tail at Scale" by Jeffrey Dean and Luiz Andre Barroso in *Comm. Of the ACM, 2013*

Distributed Computing Challenges (recap)

Scalability
- Being able to elastically scale (out and in) to meet the load demand is crucial.

Fault Tolerance
- Accept the reality that faults are common and build for quick detection and recovery.

High Availability
- Target multiple 9s availability to minimize costs for downtime.

Consistency
- Embracing **eventual consistency** for high availability is often preferred for many use-cases.

Performance
- Optimizing for tail latency is **important.**

Cloud-based application design

Design principles

The cloud revolution for application design

- The cloud changes how applications are designed

Traditional on-premises	Modern Cloud
Monolithic	Decomposed
Designed for predictable scalability	Designed for elastic scale
Relational Database	Mix of storage technologies
Synchronized processing	Asynchronous processing
Design to avoid failures	Design for failure recovery
Occasional large updates	Frequent small updates
Manual management	Automated self-management
Snowflake servers	Immutable infrastructure

- https://docs.microsoft.com/en-us/azure/architecture/guide/

Design principles for cloud applications I

- **Design for self-healing.**
  - In a distributed system, failures happen all the time. Design the application to be self-healing
  .
- **Make all things redundant.**
  - Build redundancy into your application to avoid having single points of failure.
- **Minimize coordination.**
  - Minimize coordination between application services to achieve better scalability.
- **Design to scale out.**
  - Design your application so that it can scale horizontally, adding or removing new instances on demand.
- **Partition around limits.**
  - Use partitioning to work around database, network and compute limits.

Design principles for cloud applications II

- **Use of stateless services.**
  - Scaling without having a state is trivial.

- **Caching**
  - Latency is king. Caching helps to significantly reduce the job's latency.

- **Use the best data store for the job.**
  - Pick the storage technology that is the best fit for your data and how it will be used.

- **Distribute computation**
  - Partition/Aggregate compute pattern is one that scales pretty well.

- **Design for evolution**
  - An evolutionary design is key for continuous innovation.

Designing Efficient Systems

- **Important skill**: ability to **estimate** the **performance** of a system **without** actually **building it**!

- Do **back-of-the-envelope** calculations

- e.g., **How long to generate image results page (with 30 256K-image thumbnails)?**
  - **Design 1:** read 30 images serially:
    - 30*10ms/seek + 30*256K / 30MB/s = 560ms
  - **Design 2:** issue 30 reads in parallel:
    - 10ms/seek + 256K / 30 MB/s = 18ms

- Lots of variations (caching, pre-computation, etc.)

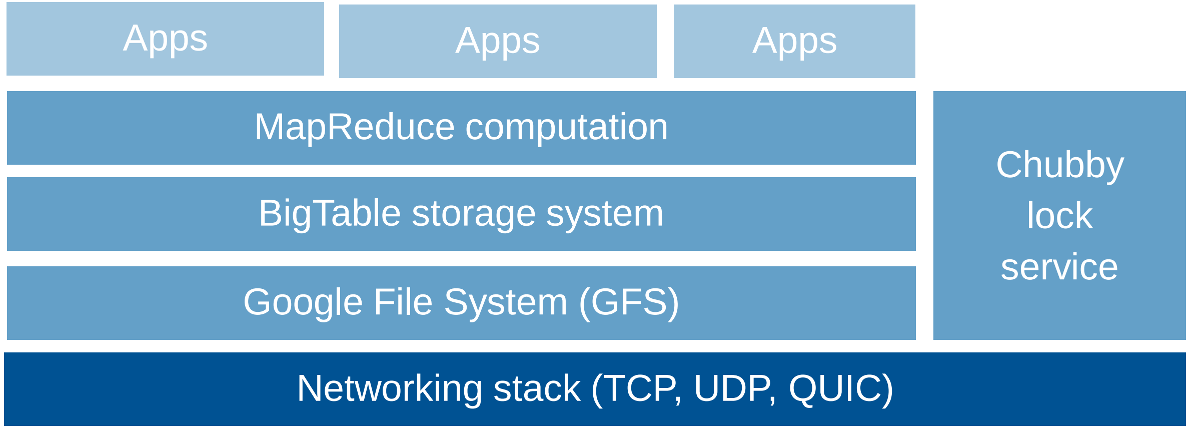
Action	Latency [ns]
L1 cache reference	0.5
Branch mis-prediction	5
L2 cache reference	7
Mutex lock/unlock	100
Main memory reference	100
Compress 1k bytes with Zippy	10’000
Send 2k bytes over 1Gbps network	20’000
Read 1MB sequentially from memory	250’000
Round trip within the same datacenter	500’000
Disk seek	10’000’000
Read 1MB sequentially from network	10’000’000
Read 1MB sequentially from disk	30’000’000
Send packet CA -> Netherlands -> CA	150’000’000

https://static.googleusercontent.com/media/research.google.com/en//people/jeff/Stanford-DL-Nov-2010.pdf

Abstractions for Scalable Systems

e.g., Google uses several **layers of abstraction**

- Runs applications (e.g., search, mail, etc.) on top of the highest level
- **Each layer** is **scalable**, **network-aware** and **fault-tolerant**



- **Know** the basic **building blocks** (e.g., language libraries, data structures, indexing systems, datastores).
  – Not just their interfaces, but **understand** their **implementation** (at least at a high level)
  – If you do not know what’s going on, you cannot do decent back-of-the-envelope calculations!

Modern Scalable Distributed Systems Stacks

- The whole spectrum is a lot more diverse, but just as a high-level overview

Applications (e.g., Gmail, Facebook, mobile apps, etc.)				
Files, dirs	put, get	lock, unlock	tasks	enq., deq.
Distributed file system (GFS, HDFS, NFS)	Distributed KV store (S3,Dynamo, Cassandra)	Distributed locking service (Chubby, ZooKeeper)	Distributed computing (Spark, MapReduce)	Message Queues (Amazon SQS)
Networking stack (TCP, UDP, QUIC)				

- Plus, many internal services for auto-scaling, monitoring, caching, security, etc.

Google/FB/Amazon System Design Interview

- Design a **scalable service**: e.g., Dropbox, Instagram, Twitter, YouTube/Netflix, etc.

- Typical steps:
  1. Find the **requirements** and **goals** of the system (e.g., **functional**, **non-functional**)
  2. Figure out the **workloads** the system should be optimized for (e.g., is it a read-heavy workload, etc.)
  3. Do a **back-of-the-envelope calculations** for estimated storage capacity needs
  4. High-level system **design**
  5. Do the **database schema** based on the **functional requirements**
  6. Do the **large-scale system design** based on the **non-functional requirements**
     – How do you **scale** the system?
     – How can you make it **reliable** and **redundant?**
     – How would you do **data sharding?**
     – Cache and **load balancing**?
  7. How can you **implement** the **functional compute** requirements in the scaled system

https://www.educative.io/courses/grokking-the-system-design-interview

Cloud-based application design

Data Infrastructure

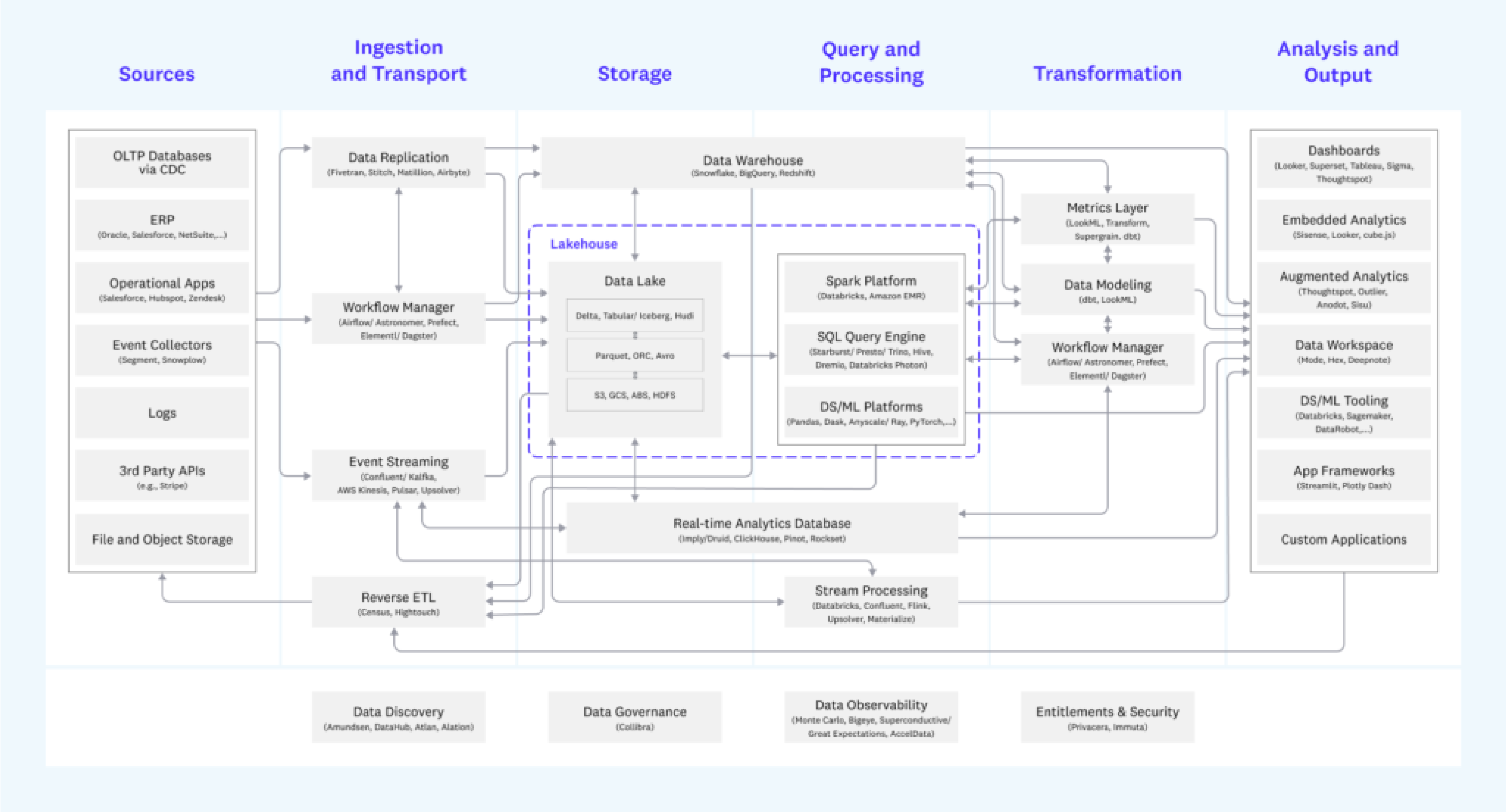
Data infrastructure for the cloud

- Need to account for the **full lifecycle** of data
  - Meet the requirements of each stage: **ingestion**, **storage**, **processing**, and **visualization**.

  - **Coordinate** the efficient **flow of data** between stages
  - **Efficient** execution of **computations** using the data.

Unified Architecture for Data Infrastructure

- Excluding transactional systems (OLTP), log processing, and SaaS analytics applications.



https://future.com/emerging-architectures-modern-data-infrastructure/

References

In addition to cross-references provided in the slides

Some material based on:

- Lecture notes by Prof. Peter Pietzuch (Imperial)
- “Software Engineering Advice for Building Large-Scale Distributed Systems” by Jeff Dean (Google)
- “Building Large-Scale Internet Services” by Jeff Dean (Google) (link)
- “Azure Application Architecture Guide” by Microsoft (link)
- “Architecting for the Cloud” by AWS (link)