# Big Data
## Session 2: Storage Infrastructures

universität koblenz
weiter:denken

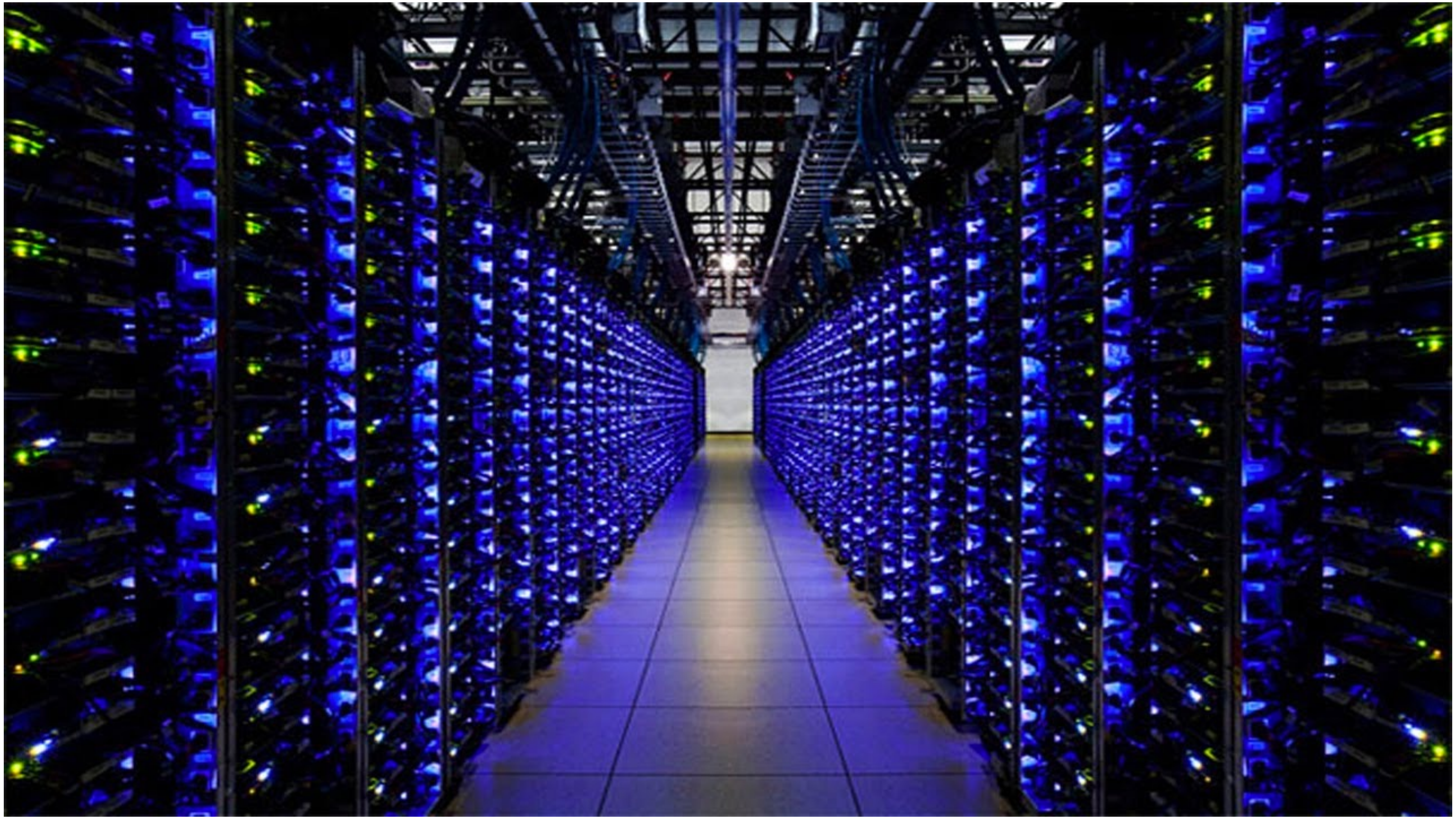
# Last week

universität koblenz
weiter:denken

- Big Data Definitions and Systems
  - Scale-up vs scale-out
  - Batch processing vs streaming
  - Hadoop, Spark
- Big Data Use-Cases
  - Internet, Retail, and more
- Introduction to Hadoop

# Big Data is processed using many different servers

universität koblenz
weiter:denken



# Intended Learning Outcomes

**At the end of this lecture, you will be able to:**

- Explain challenges of distributed IT infrastructures
- Differentiate between different distributed file systems
- Describe current solutions for distributed data storage

# Outline

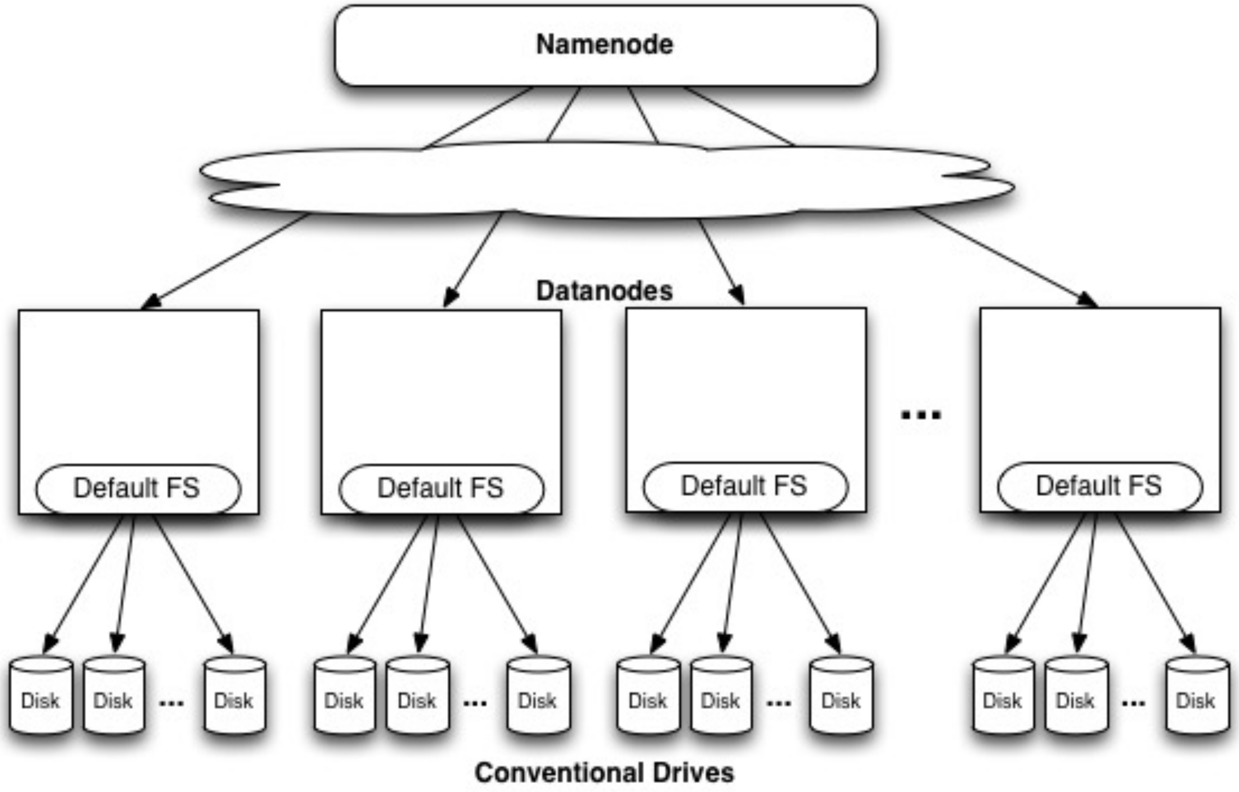
- Distributed Infrastructures
- Google File System
- HDFS
- Apache Spark RDD

# Distributed systems

[Wikipedia] A distributed system is a model in which components located on **networked computers** communicate and **coordinate their actions** by passing **messages**. The components interact with each other in order to achieve a **common goal**.

# Distributed systems

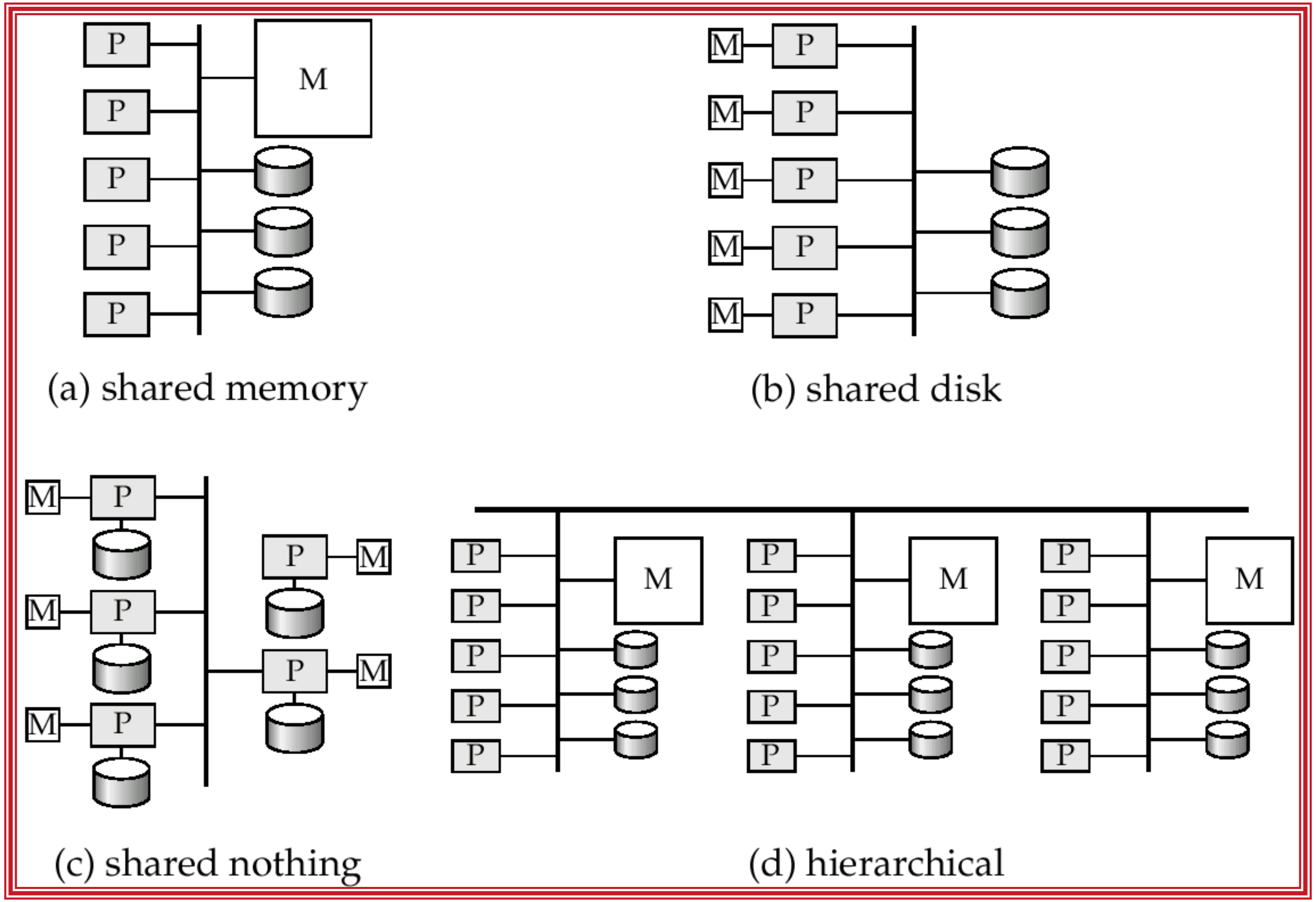
- Large Data Volumes
- Store them over a distributed system



# Databases are suitable for parallelism

- Data can be partitioned across multiple disks for parallel I/O.
- Data can be partitioned and each processor can work independently on its own partition.
- Concurrency control takes care of conflicts.

# Parallel database architectures



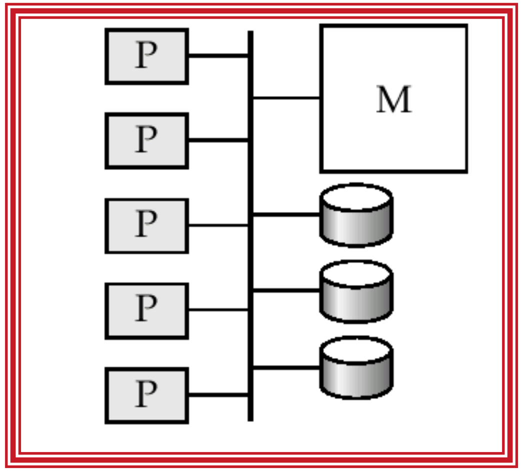
# Parallel database architectures



- **Shared memory** -- processors share a common memory
- **Shared disk** -- processors share a common disk
- **Shared nothing** -- processors share neither a common memory nor common disk
- **Hierarchical** -- hybrid of the above architectures

# Shared Memory

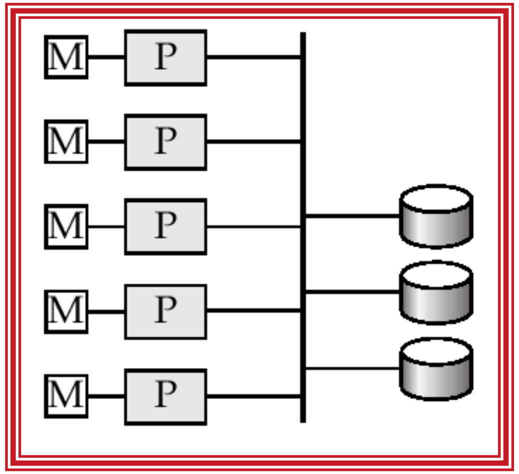
- Extremely efficient communication between processors
- Downside –is not scalable beyond 32 or 64 processors
- Widely used for lower degrees of parallelism (4 to 8).



# Shared Disk

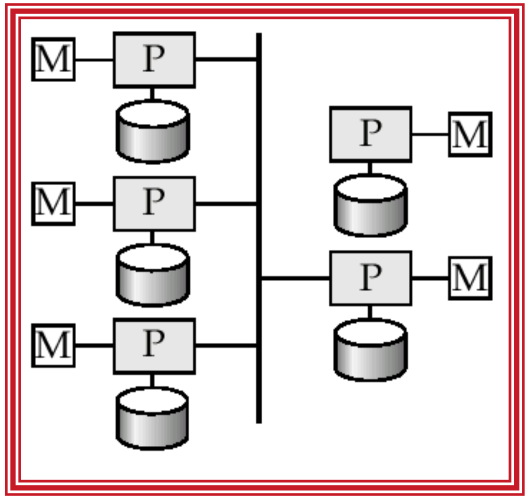
universität koblenz
weiter:denken

- Examples: IBM Sysplex and DEC clusters running Rdb were early commercial users
- Downside: bottleneck at interconnection to the disk subsystem.
- Shared-disk systems can scale to a somewhat larger number of processors, but communication between processors is slower.



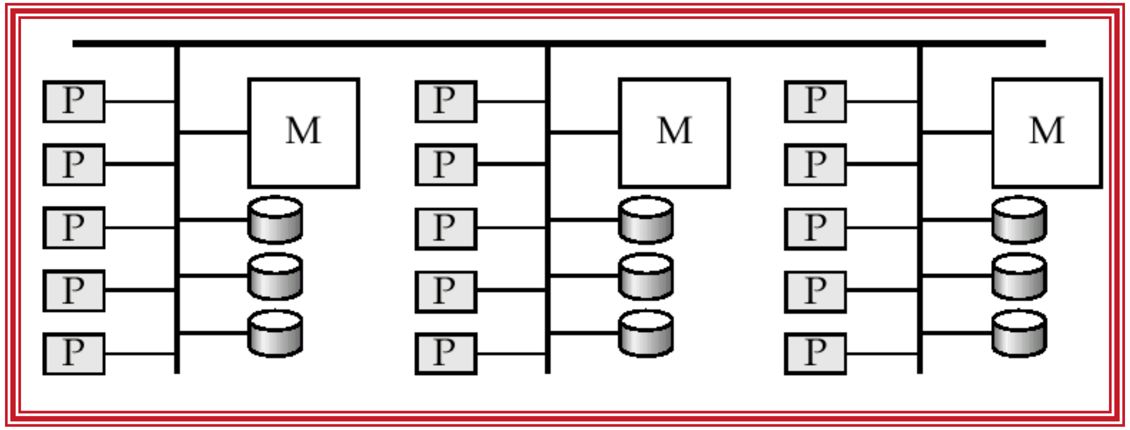
# Shared Nothing

- Data accessed from local disks (and local memory accesses) do not pass through interconnection network, thereby minimizing the interference of resource sharing.
- Shared-nothing multiprocessors can be scaled up to thousands of processors without interference.
- Main drawback: cost of communication and non-local disk access; sending data involves software interaction at both ends.



# Hierarchical

Combines characteristics of shared-memory, shared-disk, and shared-nothing architectures.



# Distributed

- Databases having storage devices distributed over a network of connected computers

- Reasons for building distributed systems:
  - sharing data
  - autonomy
  - Availability

Which of these architectures is the most scalable?

# Shared Nothing

- Most scalable architecture
  - Minimizes interference by minimizing resource sharing
  - Can use commodity hardware

- Also most difficult to program and manage
  - Processor=server=node
  - P=number of nodes

# CAP Theorem

- Conjectured by Prof. Eric Brewer at PODC (Principle of Distributed Computing) 2000 keynote talk
- Described the *trade-offs involved in distributed system*
- It is impossible for a web service to provide following *three guarantees at the same time*:
  - **Consistency**
  - **Availability**
  - **Partition-tolerance**

# CAP Theorem

Source: https://www.youtube.com/watch?v=UXes4JwUG3w

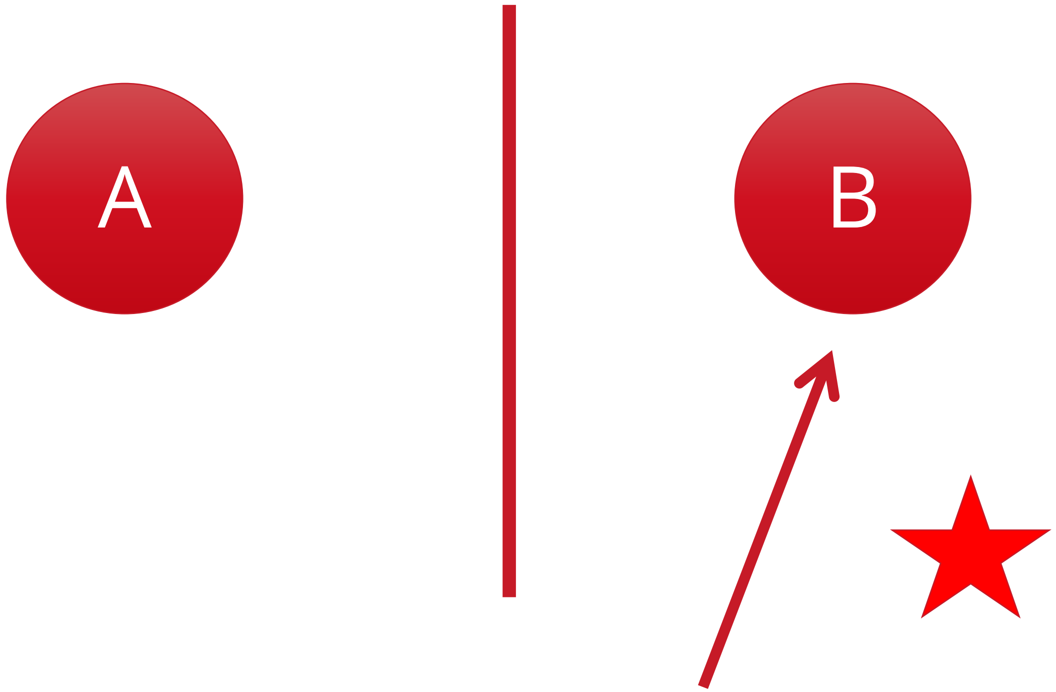
# CAP Theorem

- **Consistency** means that each client always has the same view of the data.
- **Availability** means that all clients can always read and write.
- **Partition tolerance** means that the system works well across physical network partitions.

- Only 2 out of 3 can be implemented

Details: Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." ACM SIGACT News 33.2 (2002): 51-59.

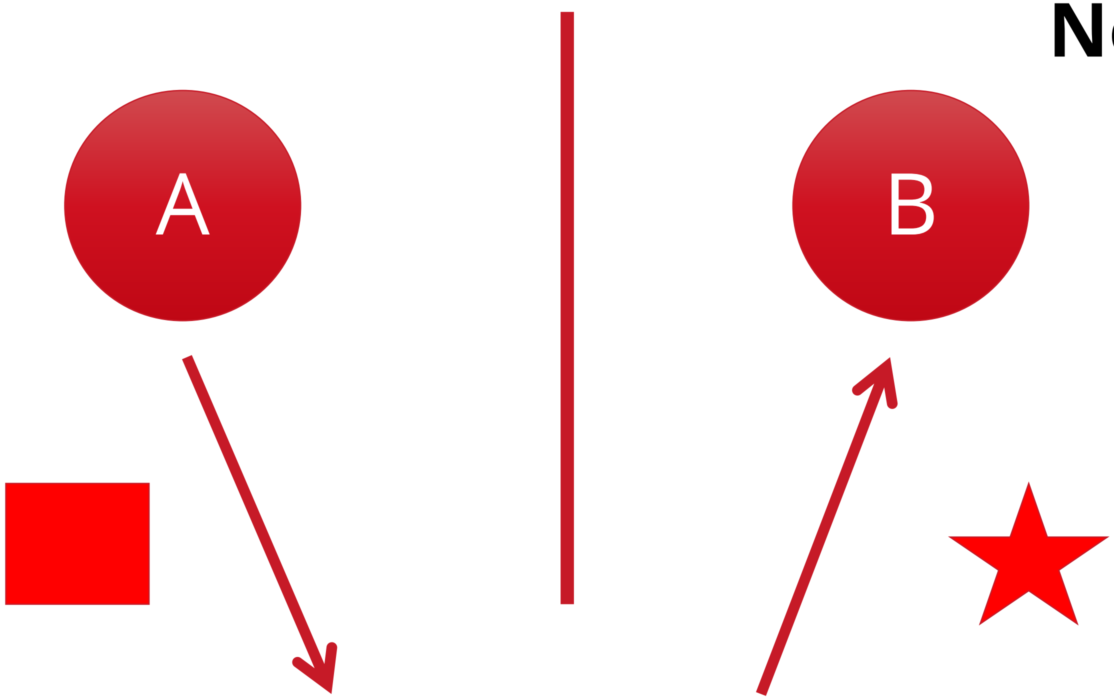
# CAP Theorem: Proof



# CAP Theorem: Proof

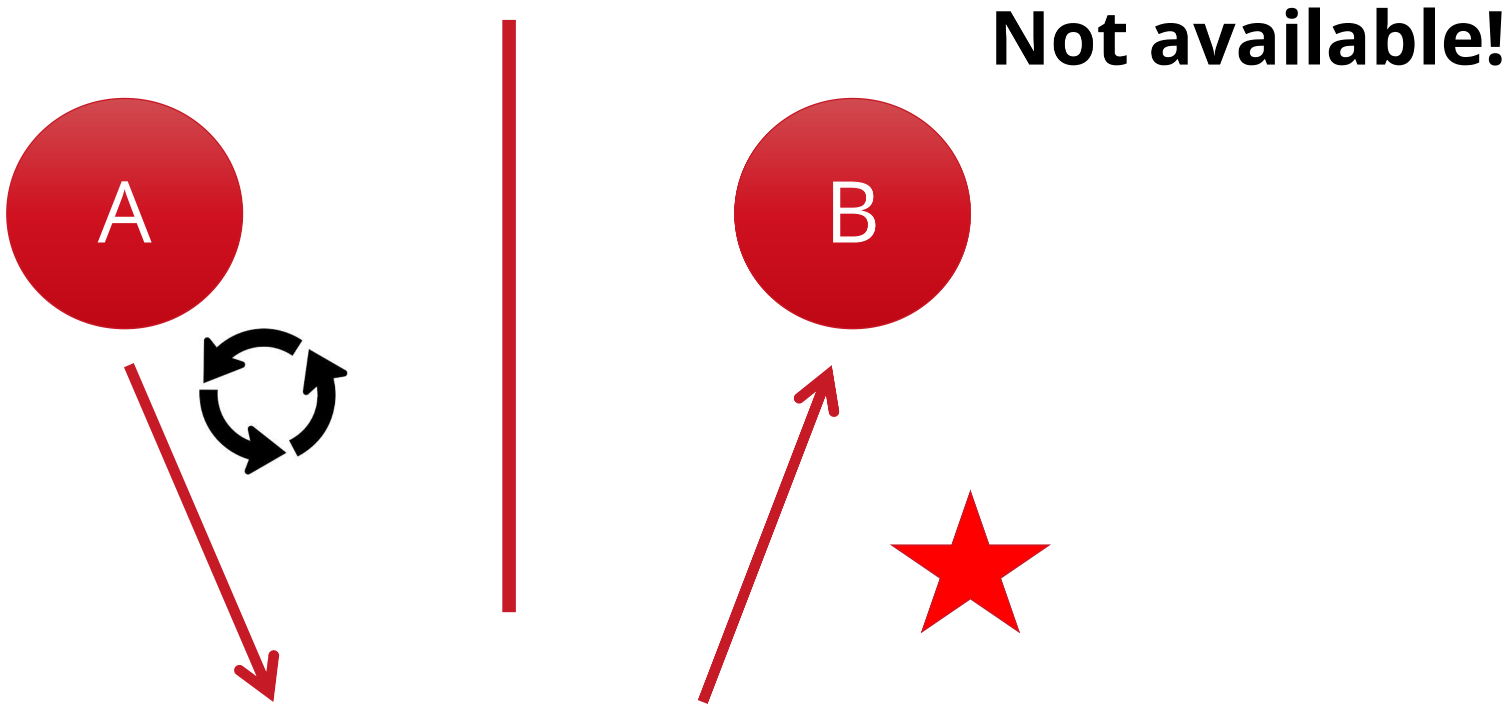


**Not consistent!**



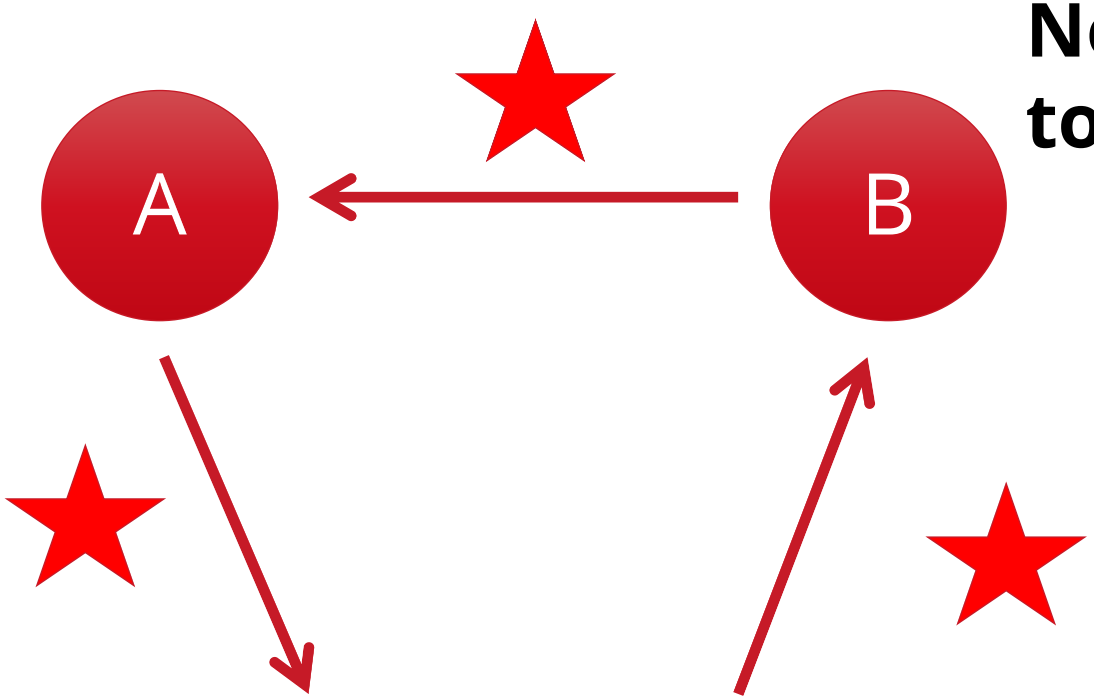

Respond to client

# CAP Theorem: Proof




Wait to be updated

# CAP Theorem: Proof




**Not partition tolerant!**

A gets updated from B

# Why is this important?

universität koblenz
weiter:denken

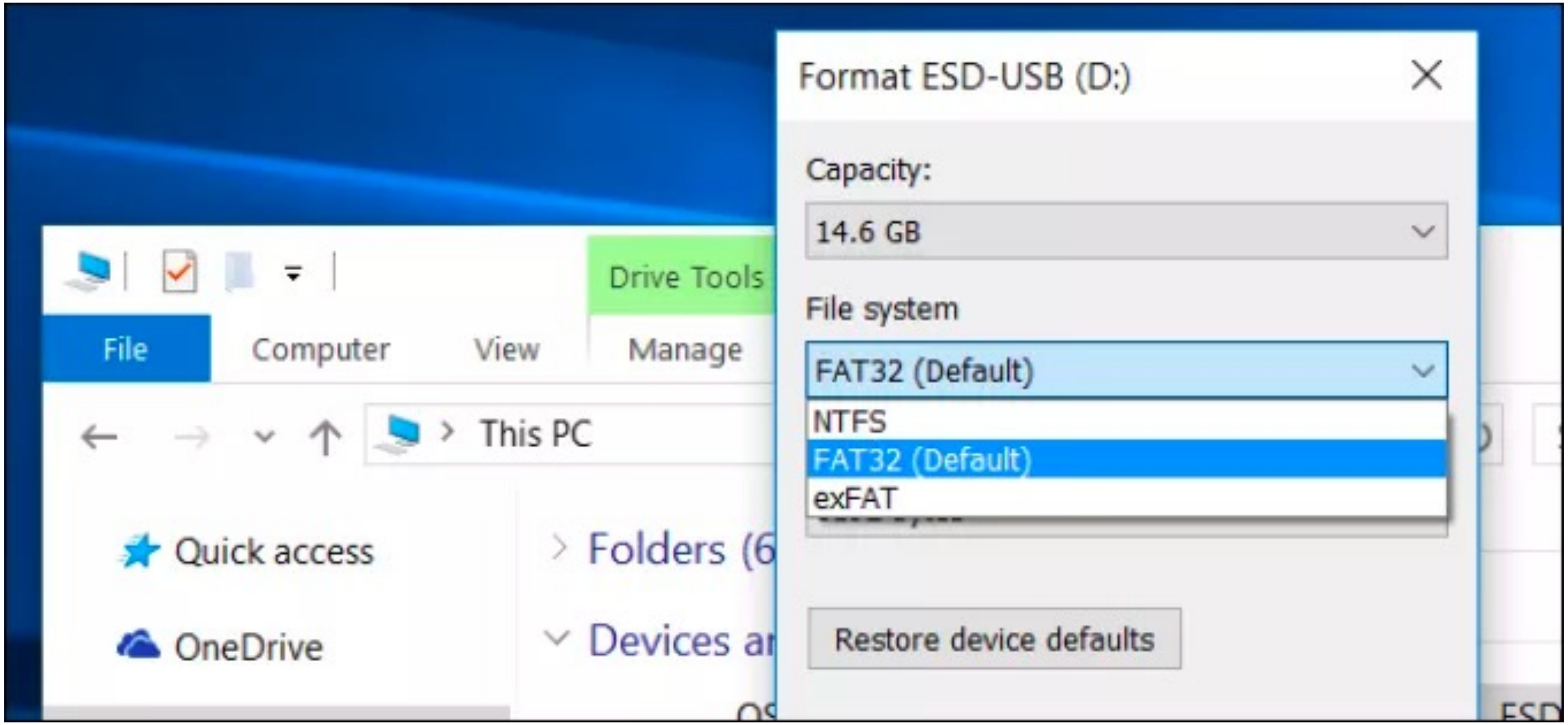
- Big data analytics is based on **distributed data storage**
- CAP theorem describes the **trade-offs** involved in distributed systems
- A proper understanding of CAP theorem is essential to **making decisions** about the future of distributed database **design**
- Misunderstanding can lead to **erroneous or inappropriate** design choices

# Outline

- Distributed Infrastructures
- **Google File System**
- HDFS
- Apache Spark RDD

What is a file system?

# File systems determine how data is stored and retrieved



# Use Case Google (in 2003)



- Huge amounts of data to store and process
    - 20+ billion web pages x 20KB/page = 400+ TB
    - Reading from one disk 30-35 MB/s
        - Four months just to read the web
        - 1000 hard drives just to store the web
        - Even more complicated if we want to process data
- **Scalable solution needed**

# Google's requirements

- A scalable distributed file system for large data-intensive applications running on inexpensive commodity hardware
- Distributed file system: Manage file storage across a network of machines
- A tool for processing large data sets in parallel on inexpensive commodity hardware
- Optimal performance

# Google File System (GFS)

universität koblenz
weiter:denken

- Scalable distributed file system for large distributed data intensive applications

- Build on Linux operation system

- It delivers high aggregate performance to a large number of clients

Details: Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, ACM, Bolton Landing, NY (2003), pp. 20-43.

# GFS Assumption

- Hardware **failures are common** (commodity hardware)
- **Files are large** (GB/TB) and their number is limited (millions, not billions)
- Two main types of reads: **large streaming reads** and **small random reads**
- Workloads with **sequential writes** that **append** data to files
- Once written, files are **seldom modified** (!=append) again
  - Random modification in files possible, but not efficient in GFS

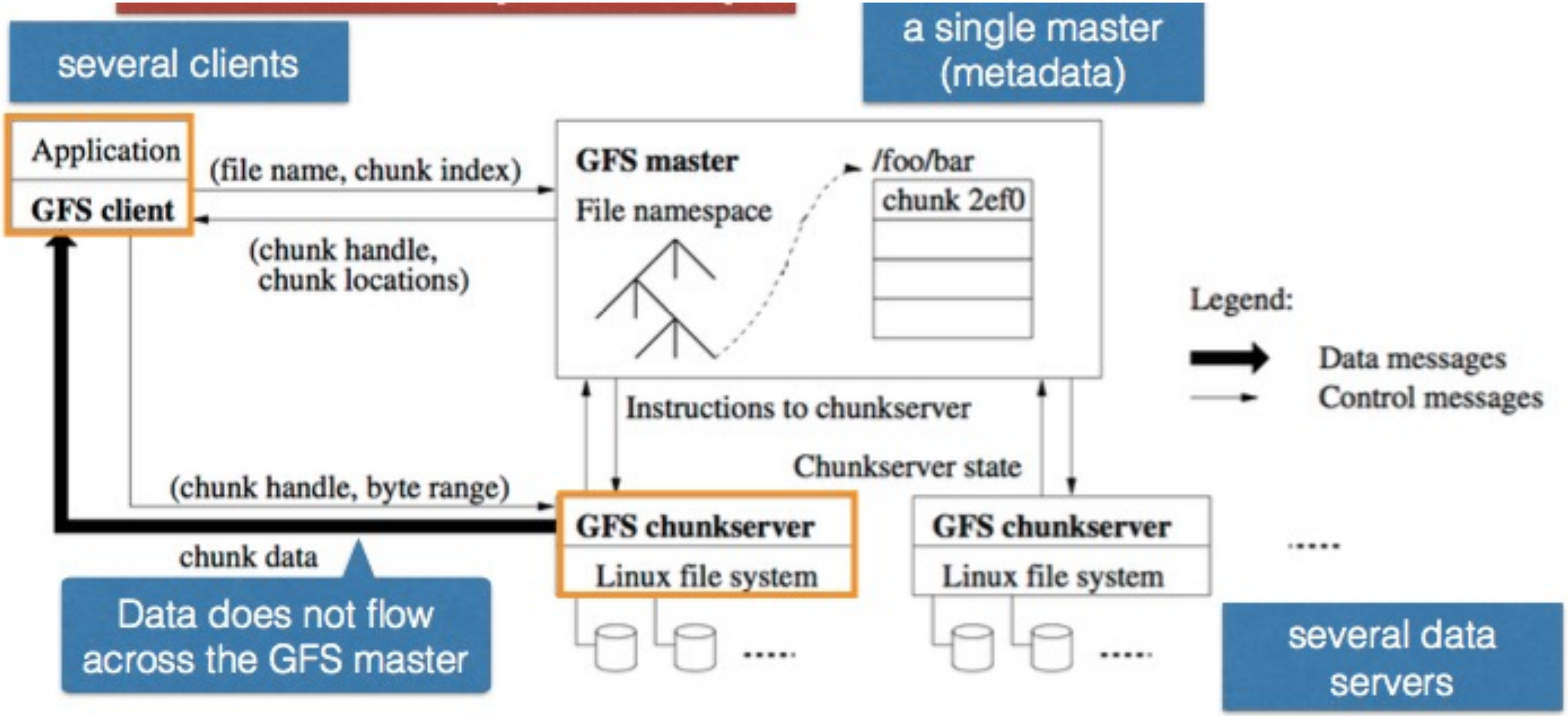
# GFS is not good for…

- **Low latency data access** (in the milliseconds range)
- **Many small files**
- **Constantly changing data**

# Files in GFS

- A **single file** can contain **many objects**
- Files are divided into **fixed size chunks** (64MB)
  - In Hadoop 128MB
- **Chunkservers** store chunks on local disk as “normal” files
- Files are **replicated** (by default 3 times) across all chunk servers
- **master** maintains all **file system metadata**
- **To read/write data**: client communicates with master (metadata)

# GFS Architecture



# Are there any limitations?

# Single master architecture

- Single master **simplifies the design** tremendously
  - Chunk placement and replication with **global knowledge**
- Single master in a large cluster can become a **bottleneck**

# Outline

- Distributed Infrastructures
- Google File System
- **HDFS**
- Apache Spark RDD

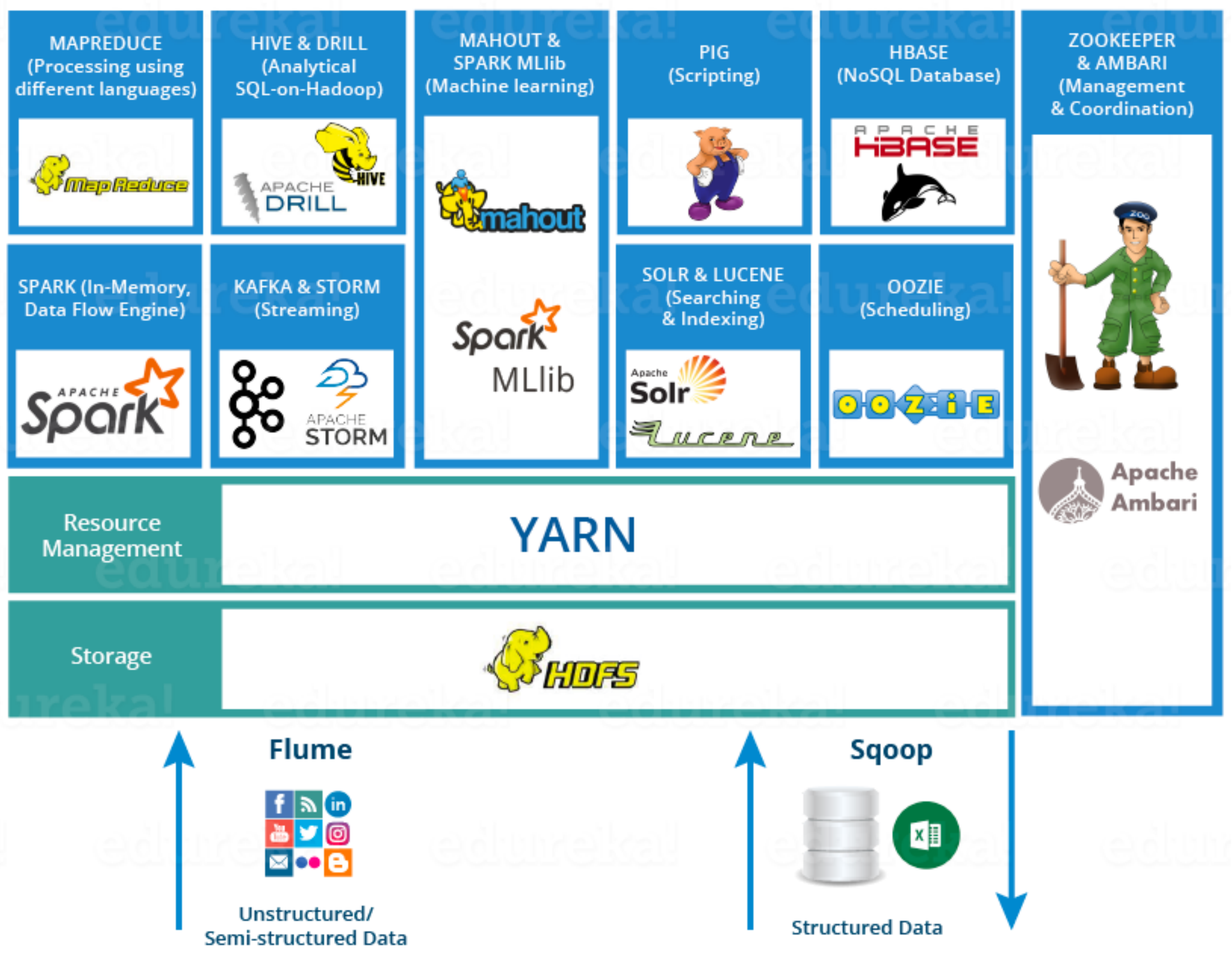
# Hadoop Distributed File System (HDFS)

universität koblenz
weiter:denken

- Inspired by Google File System
- Scalable, distributed, portable file system written in Java for Hadoop framework
- Primary distributed storage used by Hadoop applications
- HFDS can be part of a Hadoop cluster or can be a stand-alone general purpose **distributed file system**
- Reliability and **fault tolerance** ensured by replicating data across multiple hosts

Details: K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. IEEE 26th Symposium on Mass Storage Systems and Technologies, 2010

# Hadoop ecosystem



# Few facts to remember

- Distributed file system designed to run on low cost commodity hardware
- Designed for **batch processing** rather than interactive use by users
- Write-once-read-many access model for files
- Applications that run on HDFS need streaming access to their data sets
- Supports huge data volume, data divided into 64MB (default) blocks, industry practice is 128 MB
- Highly fault-tolerant, each block replicated 3 times

# GFS vs HDFS

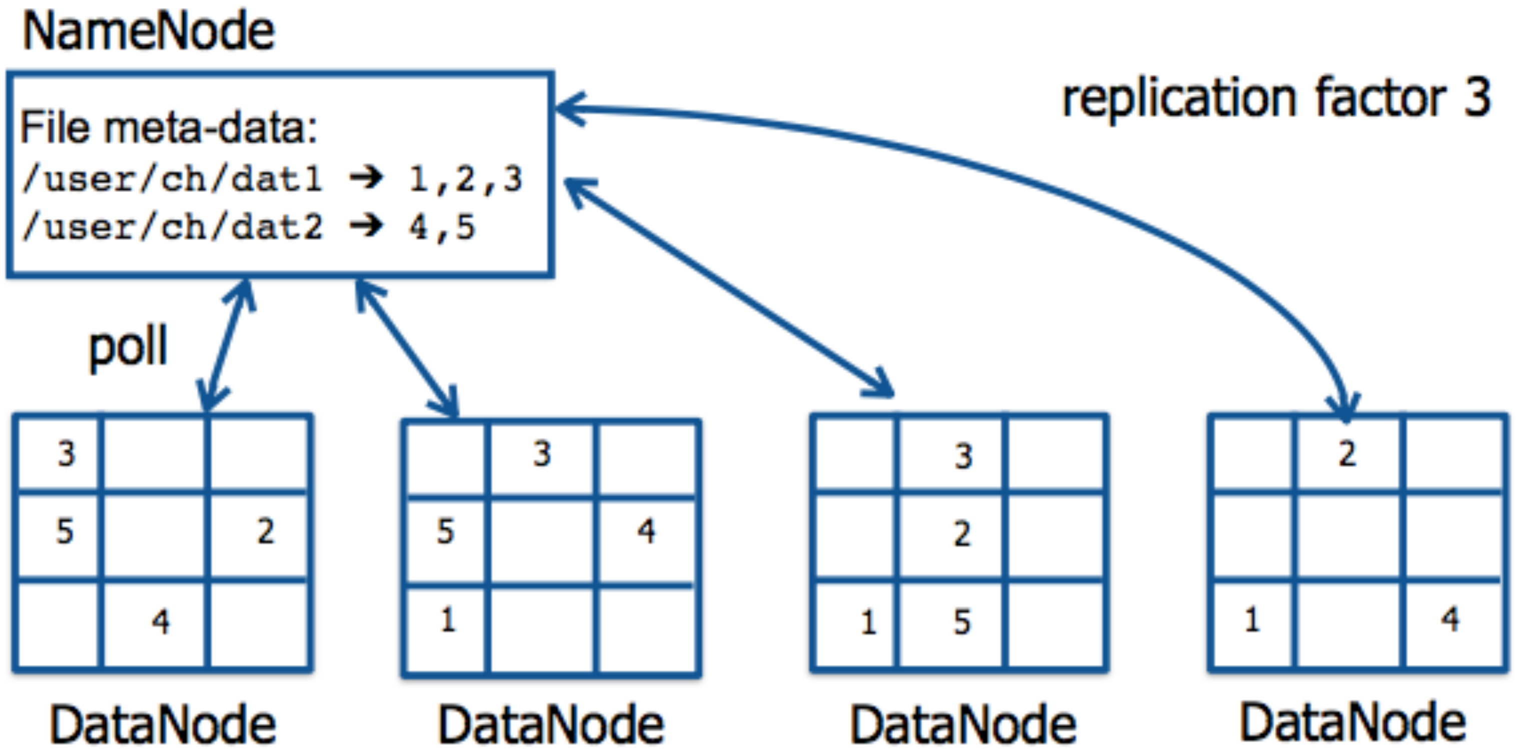
| GFS | HDFS |
|---|---|
| Master | NameNode |
| chunkserver | DataNode |
| operation log | journal, edit log |
| chunk | block |
| random file writes possible | **only append is possible** |
| multiple writer, multiple reader model | single writer, multiple reader model |
| chunk: 64KB data and 32bit checksum pieces | per HDFS block, two files created on a DataNode: data file & metadata file (checksums, timestamp) |
| default block size: 64MB | default block size: 128MB |

# HDFS

- **NameNode**
  - **Master** of HDFS, directs the slave DataNode daemons to perform low-level I/O tasks
  - Keeps track of file splitting into blocks, replication, block location, etc.

- **Secondary NameNode**: takes snapshots of the NameNode
- **DataNode**: each slave machine hosts a DataNode daemon

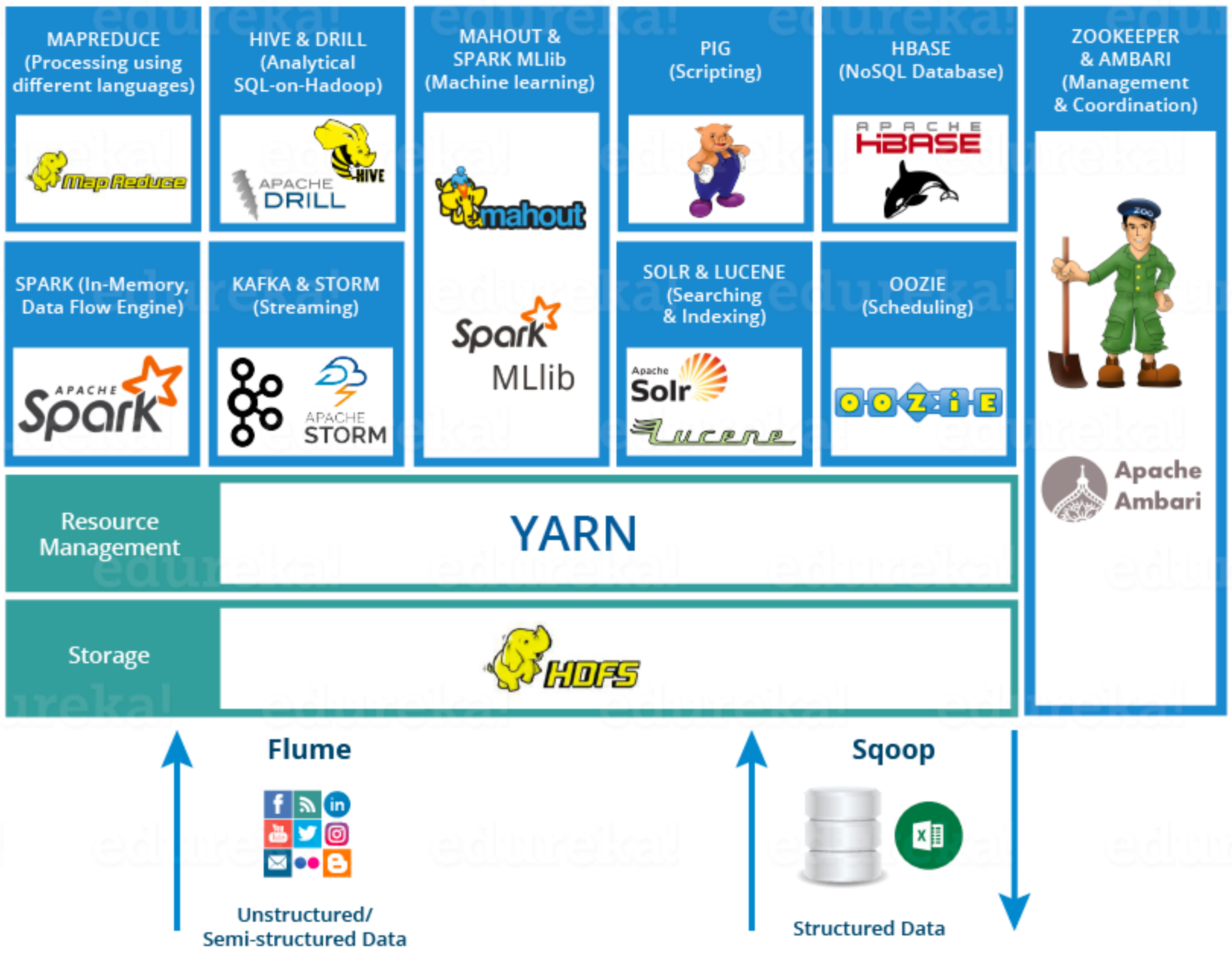
# NameNodes and DataNodes



# Outline

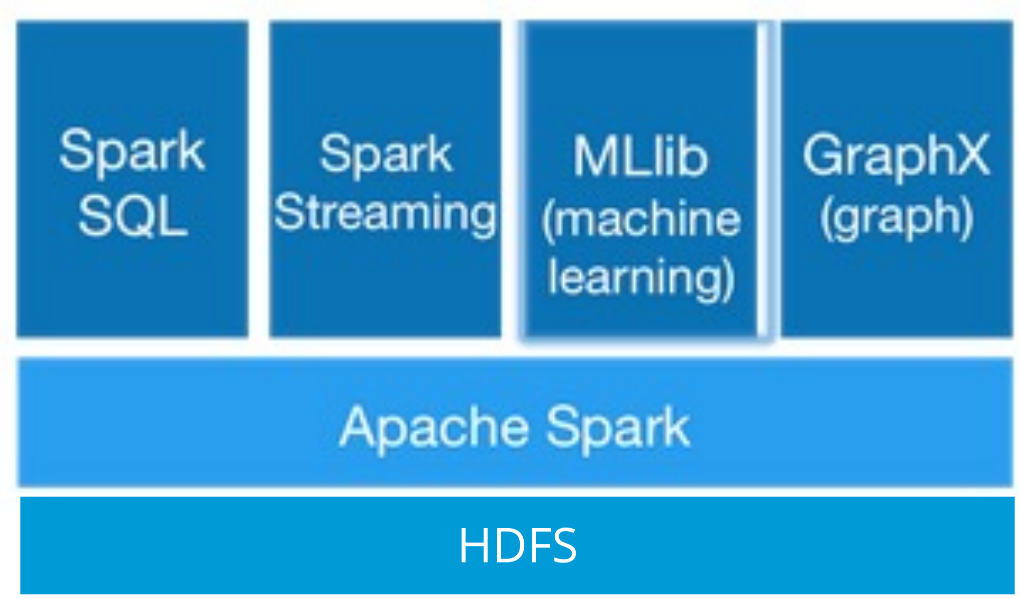
- Distributed Infrastructures
- Google File System
- HDFS
- **Apache Spark RDD**

# Hadoop ecosystem



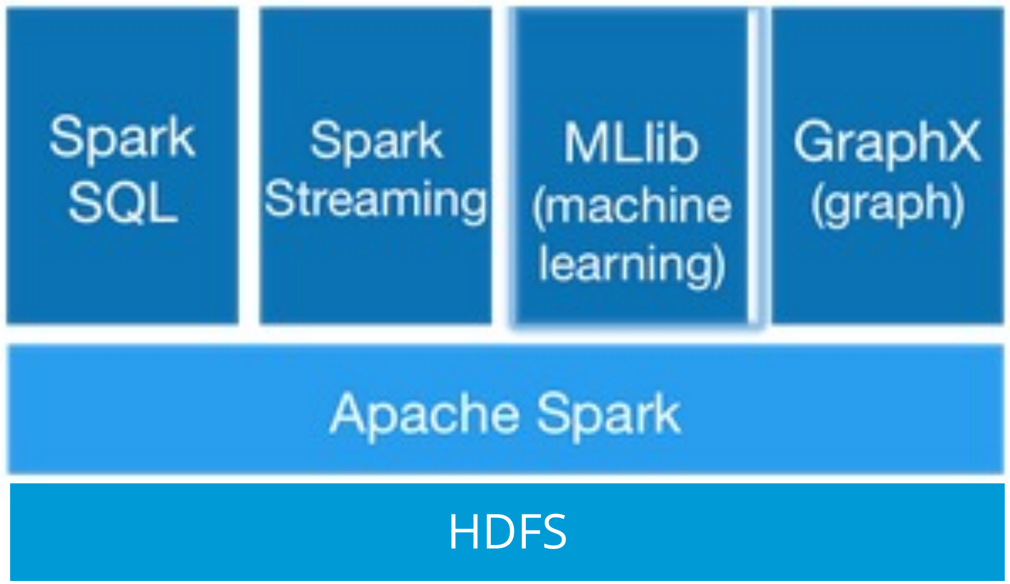
# What is Apache Spark?

- Apache Spark is a fast and general-purpose cluster computing system for large scale data processing
- Suitable for batch processing **and** real-time processing
- High-level APIs in Java, Scala, Python, and R



# Standard libraries

- Spark SQL
  - Allows to query structured data
- Spark Streaming
  - Intended for real-time processing.
- MLlib
  - Suitable for machine learning tasks
- GraphX
  - Graph processing




Details: M. Zaharia et al. Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM*, 59(11):56-65, 2016

# Gray sort competition: Winner Spark-based



| | Hadoop MR Record | Spark Record (2014) |
|---|---|---|
| Data Size | 102.5 TB | 100 TB |
| Elapsed Time | 72 mins | 23 mins |
| # Nodes | 2100 | 206 |
| # Cores | 50400 physical | 6592 virtualiz |
| Cluster disk throughput | 3150 GB/s (est.) | 618 GB/s |
| Network | dedicated data center, 10Gbps | virtualized (EC2) 10Gbps network |
| **Sort rate** | **1.42 TB/min** | **4.27 TB/min** |
| **Sort rate/node** | **0.67 GB/min** | **20.7 GB/min** |

Spark-based System 3x faster with 1/10 # of nodes

Sort benchmark, Daytona Gray: sort of 100 TB of data (1 trillion records)

http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

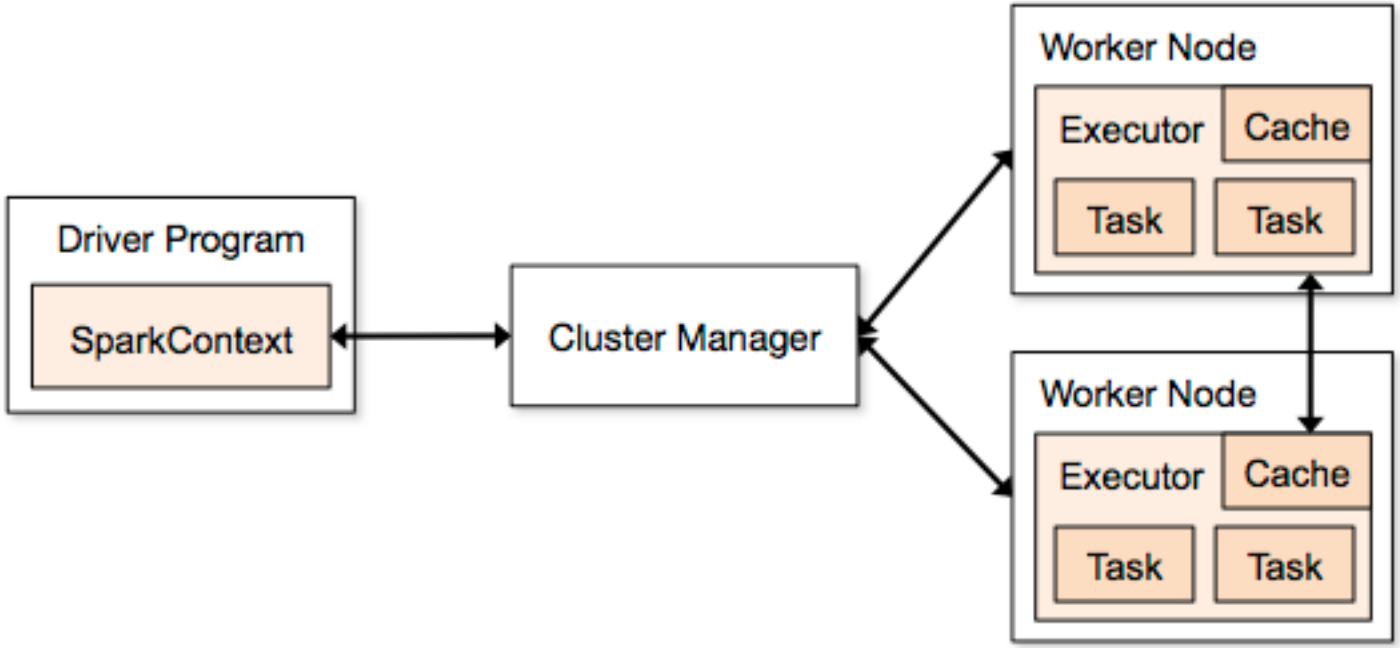
# Spark vs. Hadoop MapReduce

universität koblenz
weiter:denken

- Performance: Spark normally faster but with caveats
  - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
  - Spark generally outperforms MapReduce, but it often needs lots of memory to do well; if there are other resource-demanding services or can't fit in memory, Spark degrades
  - MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for
- Ease of use: Spark is easier to program
- Data processing: Spark more general
- Maturity: Spark maturing, Hadoop MapReduce mature

"Spark vs. Hadoop MapReduce" by Saggi Neumann (November 24, 2014)
https://www.xplenty.com/blog/2014/11/apache-spark-vs-hadoop-mapreduce/

# Spark Components



# Spark Abstractions & Concepts

- RDD: Resilient Distributed Dataset
  - Management and processing of distributed data
- DAG: Direct Acyclic Graph
  - Constructs execution flow by constructing graph comprising of nodes and edges
- SparkContext
  - Manages orchestration within a Spark cluster
- Transformations
  - "Creation of new RDDs", e.g., by filtering data
- Actions
  - Operations that return something other than an RDD

# Resilient Distributed Dataset (RDD) – key Spark construct

- RDDs represent data or transformations on data
- RDDs can be created from Hadoop InputFormats (such as HDFS files), “parallelize()” datasets, or by transforming other RDDs (you can stack RDDs)
- Actions can be applied to RDDs; actions force calculations and return values
- Lazy evaluation: Nothing computed until an action requires it
- RDDs are best suited for applications that apply the same operation to all elements of a dataset
  - Less suitable for applications that make asynchronous fine-grained updates to shared state

# Spark RDDs


Source: https://www.youtube.com/watch?v=NRo8TluH7KI

# Spark RDDs (Ch.3, Spark book)

- Based on HDFS

- Resilient distributed dataset (RDD)
  - Distributed collection of elements
  - Immutable
  - Lazy evaluation
- Spark distributes data across the cluster
  - Partitions
- Operations
  - Transformations
  - Actions

In Python: https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD

# Transformations

- Generates a new RDD partition
- Lazy evaluation (computed only when needed)
- Map, Filter, flatMap, Sample, Union, Intersection, Distinct, groupByKey, reduceByKey, sortByKey, Join, Cogroup, cartesian

# Transformations

universität koblenz
weiter:denken

- **Map(function):** Return a new distributed dataset formed by passing each element of the source through a function

- **FlatMap(function):** each input item can be mapped to 0 or more output items (instead of one as for map)

- **reduceByKey(function):** When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function, which must be of type (V,V) => V.

https://spark.apache.org/docs/latest/programming-guide.html#working-with-key-value-pairs

# Actions

- Return a final value
- Force the evaluation of transformation operations

- Reduce, Collect, Count, First, Take, takeSample, saveAsTextFile, foreach

- **Collect():** Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.

# Summary

- Distributed Infrastructures
- Google File System
- HDFS
- Apache Spark RDD

# What’s next – Column stores and Coordination

- Part 1:
  - Column stores
  - Big Table
  - HBase and Hive

- Part 2:
  - Job Scheduling
  - Coordination