SCIENCE
PASSION
TECHNOLOGY

Data Management
11 Distributed Storage & Analysis

Matthias Boehm

Graz University of Technology, Austria
Computer Science and Biomedical Engineering
Institute of Interactive Systems and Data Science
BMVIT endowed chair for Data Management

Last update: May 25, 2020

Announcements/Org

- **#1 Video Recording**
  - Link in **TeachCenter** & **TUbe** (lectures will be public)
  - **Live Streaming** Mo 4.10pm until end of semester (June 30)
  - **Office hours:** Mo 1pm-2pm (https://tugraz.webex.com/meet/m.boehm)

- **#2 Exercises**
  - **Exercise 1 graded**, feedback in TC (**plagiarism**, discussion **issues**)
  - **Exercise 2/3 in progress of being graded**
  - Exercise 4 published, deadline **June 16 11.59pm**

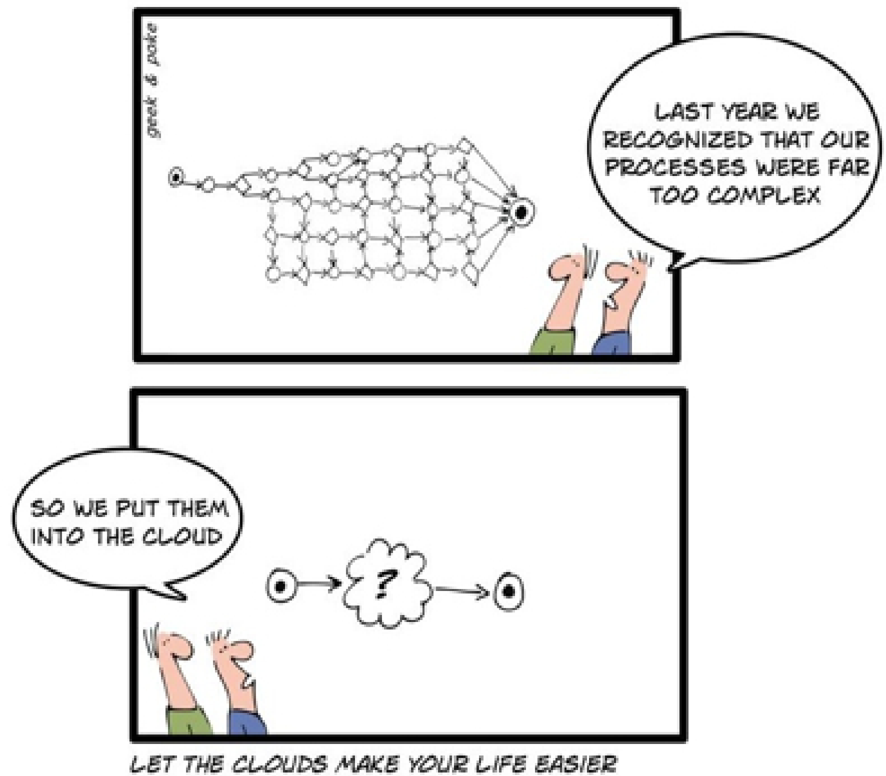
- **#3 Exam Dates** (VR Teaching Planning until June 27)
  - **June 22:** 8am-10am, **11am-1pm**, **2pm-4pm**, **5pm-7pm** (concurrently in i7, i11, i12, i13)
  - Counter-proposal: cut 8am, add June 23 6pm
  - **Deregistration possible w/o failed attempt** (even for KU/VUs)

?

Agenda

- **Cloud Computing Overview**
- **Distributed Data Storage**
- **Distributed Data Analysis**
- **Exercise 4: Large-Scale Data Analysis**

Data Integration and Large-Scale Analysis (DIA)
(bachelor/master)



Cloud Computing Overview

Motivation Cloud Computing

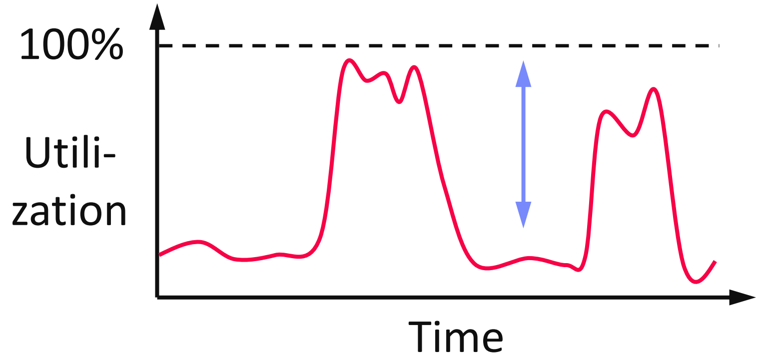
- **Definition Cloud Computing**
  - **On-demand, remote storage and compute resources, or services**
  - **User:** computing as a utility (similar to energy, water, internet services)
  - **Cloud provider:** computation in data centers / multi-tenancy

- **Service Models**
  - **IaaS: Infrastructure as a service** (e.g., storage/compute nodes)
  - **PaaS: Platform as a service** (e.g., distributed systems/frameworks)
  - **SaaS: Software as a Service** (e.g., email, databases, office, github)

→ **Transforming IT Industry/Landscape**
  - Since ~2010 increasing move from on-prem to cloud resources
  - System software licenses become increasingly irrelevant
  - Few cloud providers dominate IaaS/PaaS/SaaS markets (w/ 2018 revenue): **Microsoft Azure Cloud** ($ 32.2B), **Amazon AWS** ($ 25.7B), **Google Cloud** (N/A), **IBM Cloud** ($ 19.2B), **Oracle Cloud** ($ 5.3B), **Alibaba Cloud** ($ 2.1B)

Motivation Cloud Computing, cont.

- **Argument #1: Pay as you go**
  - No upfront cost for infrastructure
  - Variable utilization ➔ over-provisioning
  - **Pay per use or acquired resources**



- **Argument #2: Economies of Scale**
  - Purchasing and managing IT infrastructure at scale ➔ **lower cost** (applies to both HW resources and IT infrastructure/system experts)
  - Focus on **scale-out on commodity HW** over scale-up ➔ **lower cost**

- **Argument #3: Elasticity**
  - Assuming perfect scalability, work done in **constant time * resources**
  - Given virtually unlimited resources allows to reduce time as necessary

100 days @ 1 node

≈

1 day @ 100 nodes

(but beware Amdahl's law: max speedup **sp = 1/s**)

Characteristics and Deployment Models

- **Extended Definition**
  - ANSI recommended definitions for service types, characteristics, deployment models

[Peter Mell and Timothy Grance: The NIST Definition of Cloud Computing, **NIST 2011**]

- **Characteristics**
  - **On-demand self service:** unilateral resource provision
  - **Broad network access:** network accessibility
  - **Resource pooling:** resource virtualization / multi-tenancy
  - **Rapid elasticity:** scale out/in on demand
  - **Measured service:** utilization monitoring/reporting
- **Deployment Models**
  - **Public cloud:** general public, on premise of cloud provider
  - **Hybrid cloud:** combination of two or more of the above
  - **Community cloud:** single community (one or more orgs)
  - **Private cloud:** single org, on/off premises

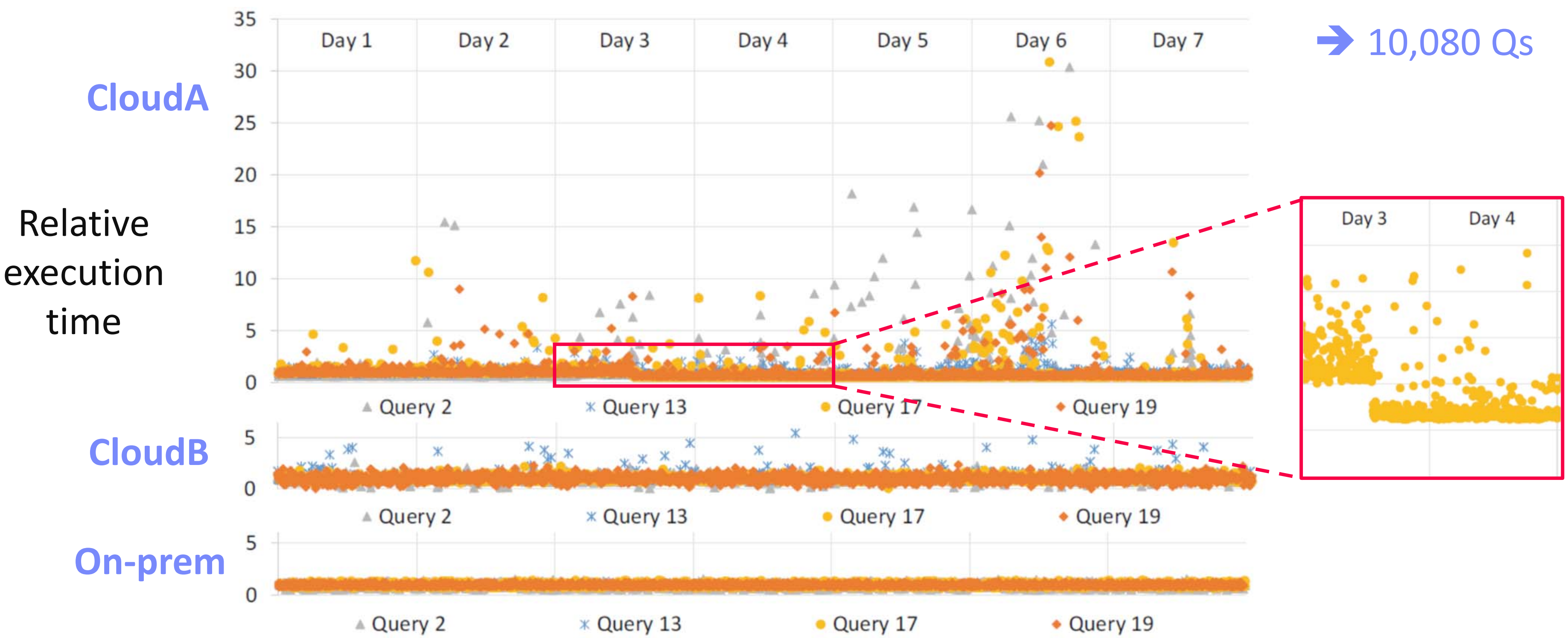
MS Azure Private Cloud

IBM Cloud Private

Excursus: 1 Query/Minute for 1 Week

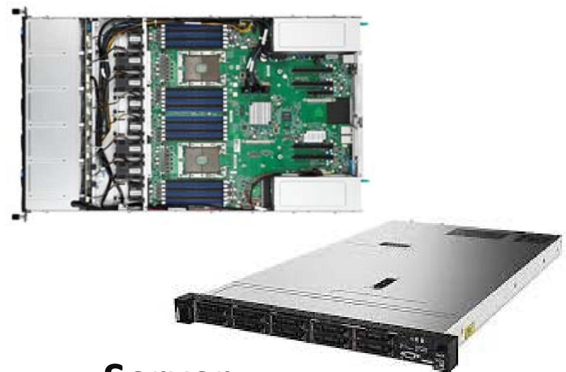
- **Experimental Setup**
  - 1GB TPC-H database, 4 queries on 2 cloud DBs / 1 on-prem DB

[Tim Kiefer, Hendrik Schön, Dirk Habich, Wolfgang Lehner: **A Query, a Minute**: Evaluating Performance Isolation in Cloud Databases. TPCTC 2014]

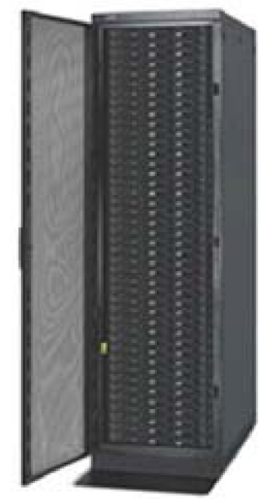


Anatomy of a Data Center

Commodity CPU:
Xeon E5-2440: 6/12 cores
Xeon Gold 6148: 20/40 cores



Server:
Multiple sockets,
RAM, disks



Rack:
16-64 servers +
top-of-rack switch

Cluster:
Multiple racks + cluster switch

Data Center:
>100,000 servers

[Google Data Center, Eemshaven, Netherlands]

Fault Tolerance

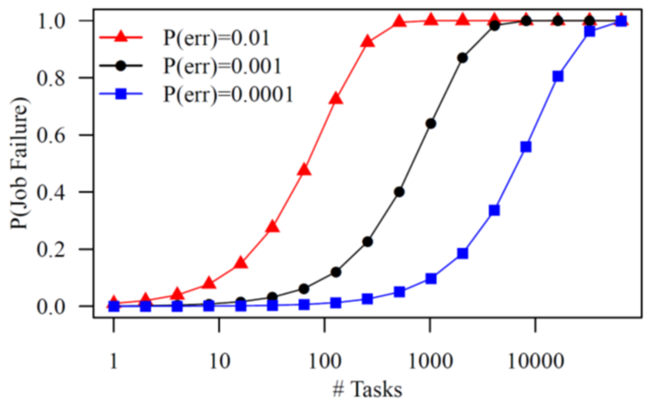
[Christos Kozyrakis and Matei Zaharia: CS349D: Cloud Computing Technology, lecture, **Stanford 2018**]

- **Yearly Data Center Failures**
  - ~0.5 overheating (power down most machines in <5 mins, ~1-2 days)
  - ~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hrs)
  - ~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hrs)
  - ~1 network rewiring (rolling ~5% of machines down over 2-day span)
  - ~20 rack failures (40-80 machines instantly disappear, 1-6 hrs)
  - ~5 racks go wonky (40-80 machines see 50% packet loss)
  - ~8 network maintenances (~30-minute random connectivity losses)
  - ~12 router reloads (takes out DNS and external vIPs for a couple minutes)
  - ~3 router failures (immediately pull traffic for an hour)
  - ~dozens of minor 30-second blips for dns
  - ~1000 individual machine failures (2-4% failure rate, at least twice)
  - ~thousands of hard drive failures (1-5% of all disks will die)

Fault Tolerance, cont.

- **Other Common Issues**
  - **Configuration issues**, partial SW updates, SW bugs
  - **Transient errors:** no space left on device, memory corruption, stragglers

- **Recap: Error Rates at Scale**
  - Cost-effective commodity hardware
  - Error rate increases with increasing scale
  - Fault Tolerance for distributed/cloud storage and data analysis



➔ Cost-effective Fault Tolerance

- **BASE** (basically **available**, soft state, **eventual consistency**)
- Effective techniques
  - ECC (error correction codes), CRC (cyclic redundancy check) for detection
  - **Resilient storage:** replication/erasure coding, checkpointing, and lineage
  - **Resilient compute:** task re-execution / speculative execution

Containerization

- **Docker Containers**
  - **Shipping container analogy**
    - Arbitrary, self-contained goods, standardized units
    - Containers reduced loading times ➔ efficient international trade
  - **#1 Self-contained package** of necessary SW and data (read-only image)
  - **#2 Lightweight virtualization** w/ shared OS and resource isolation via **cgroups**

- **Cluster Schedulers**
  - Container orchestration: scheduling, deployment, and management
  - Resource negotiation with clients
  - Typical resource bundles (CPU, memory, device)
  - Examples: **Kubernetes**, **Mesos**, (**YARN**), **Amazon ECS**, **Microsoft ACS**, **Docker Swarm**

[Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, John Wilkes: Borg, Omega, and Kubernetes. **CACM 2016**]

Borg, Omega, and Kubernetes

➔ **from machine- to application-oriented scheduling**

Example Amazon Services – Pricing (current gen)

- **Amazon EC2 (Elastic Compute Cloud)**
  - IaaS offering of different node types and generations
  - **On-demand**, **reserved**, and **spot** instances

	vCores		Mem		
m4.large	2	6.5	8 GiB	EBS Only	$0.12 per Hour
m4.xlarge	4	13	16 GiB	EBS Only	$0.24 per Hour
m4.2xlarge	8	26	32 GiB	EBS Only	$0.48 per Hour
m4.4xlarge	16	53.5	64 GiB	EBS Only	$0.96 per Hour
m4.10xlarge	40	124.5	160 GiB	EBS Only	$2.40 per Hour
m4.16xlarge	64	188	256 GiB	EBS Only	$3.84 per Hour

- **Amazon ECS (Elastic Container Service)**
  - PaaS offering for Docker containers
  - Automatic setup of Docker environment

Pricing according to EC2
(in EC2 launch mode)

- **Amazon EMR (Elastic Map Reduce)**
  - PaaS offering for Hadoop workloads
  - Automatic setup of YARN, HDFS, and specialized frameworks like Spark
  - **Prices in addition to EC2 prices**

m4.large	$0.117 per Hour	$0.03 per Hour
m4.xlarge	$0.234 per Hour	$0.06 per Hour
m4.2xlarge	$0.468 per Hour	$0.12 per Hour
m4.4xlarge	$0.936 per Hour	$0.24 per Hour
m4.10xlarge	$2.34 per Hour	$0.27 per Hour
m4.16xlarge	$3.744 per Hour	$0.27 per Hour

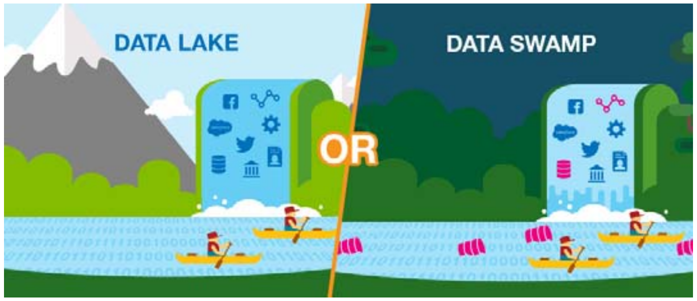
Distributed Data Storage

Cloud Object Storage

Distributed File Systems

Data Lakes

- **Concept "Data Lake"**
  - **Store massive amounts of un/semi-structured, and structured data** (append only, no update in place)
  - **No need for architected schema** or upfront costs (unknown analysis)
  - Typically: file storage in open, raw formats (inputs and intermediates)
  - ➔ **Distributed storage and analytics** for scalability and agility
- **Criticism: Data Swamp**
  - Low data quality (lack of schema, integrity constraints, validation)
  - Missing meta data (context) and data catalog for search
  - ➔ **Requires proper data curation / tools** According to priorities (data governance)



[**Credit:** www.collibra.com]

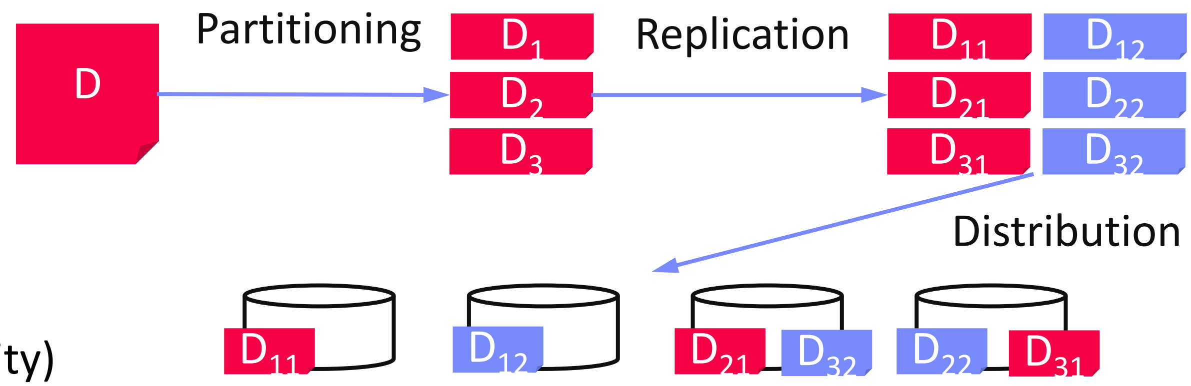
- **Excursus: Research Data Management**
  - FAIR data principles: findable, accessible, interoperable, re-usable

Object Storage

- **Recap: Key-Value Stores**
  - **Key**-value mapping, where values can be of a variety of data types
  - APIs for CRUD operations; scalability via sharding (**objects** or object segments)

- **Object Store**
  - Similar to key-value stores, but: **optimized for large objects in GBs and TBs**
  - Object identifier (**key**), **meta data**, and object as binary large object (**BLOB**)
  - APIs: often REST APIs, SDKs, sometimes implementation of DFS APIs

- **Key Techniques**
  - Partitioning
  - Replication & Distribution
  - Erasure Coding (partitioning + parity)



Object Storage, cont.

- **Example Object Stores / Protocols**
  - Amazon Simple Storage Service (S3)
  - OpenStack Object Storage (Swift)
  - IBM Object Storage
  - Microsoft Azure Blob Storage

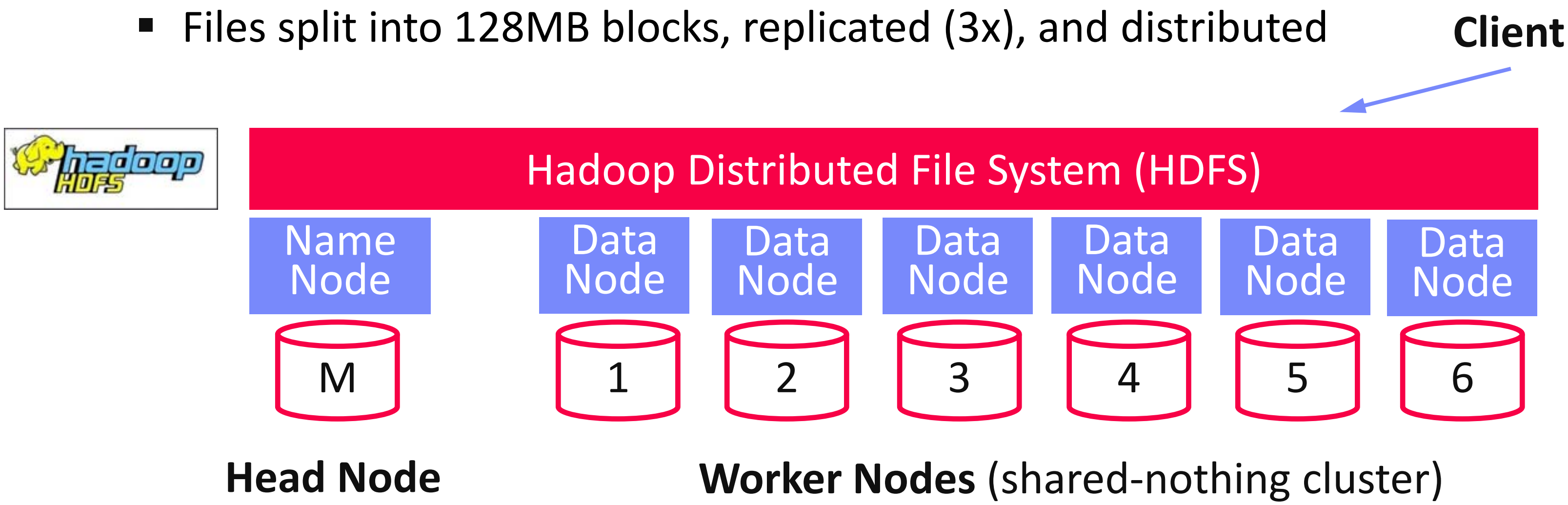
- **Amazon S3**
  - Reliable object store for photos, videos, documents or any binary data
  - **Bucket:** Uniquely named, static data container
    `http://s3.aws-eu-central-1.amazonaws.com/mboehm-b1`
  - **Object:** key, version ID, value, metadata, access control
  - Single (5GB)/multi-part (5TB) upload and direct/BitTorrent download
  - **Storage classes:** STANDARD, STANDARD_IA, GLACIER, DEEP_ARCHIVE
  - **Operations:** GET/PUT/LIST/DEL, and SQL over CSV/JSON objects

Hadoop Distributed File System (HDFS)

- **Brief Hadoop History**
  - Google's GFS + MapReduce [ODSI'04]
    → Apache Hadoop (2006)
  - Apache Hive (SQL), Pig (ETL), Mahout/SystemML (ML), Giraph (Graph)

[Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: The Google file system. **SOSP 2003**]

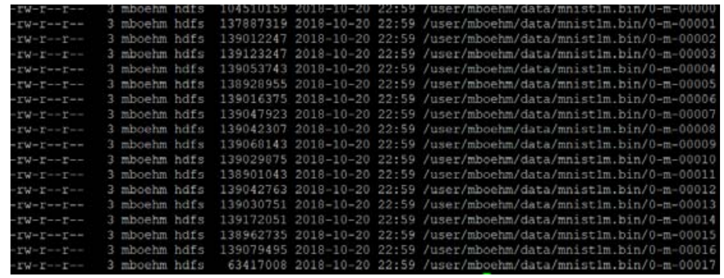
- **HDFS Overview**
  - Hadoop's distributed file system, for large clusters and datasets
  - Implemented in Java, w/ native libraries for compression, I/O, CRC32
  - Files split into 128MB blocks, replicated (3x), and distributed



Hadoop Distributed File System, cont.

- **HDFS NameNode**
  - Master daemon that manages file system namespace and access by clients
  - Metadata for all files (e.g., replication, permissions, sizes, block ids, etc)
  - **FSImage:** checkpoint of FS namespace
  - **EditLog:** **write-ahead-log (WAL)** of file write operations (merged on startup)

`hadoop fs -ls ./data/mnist1m.bin`

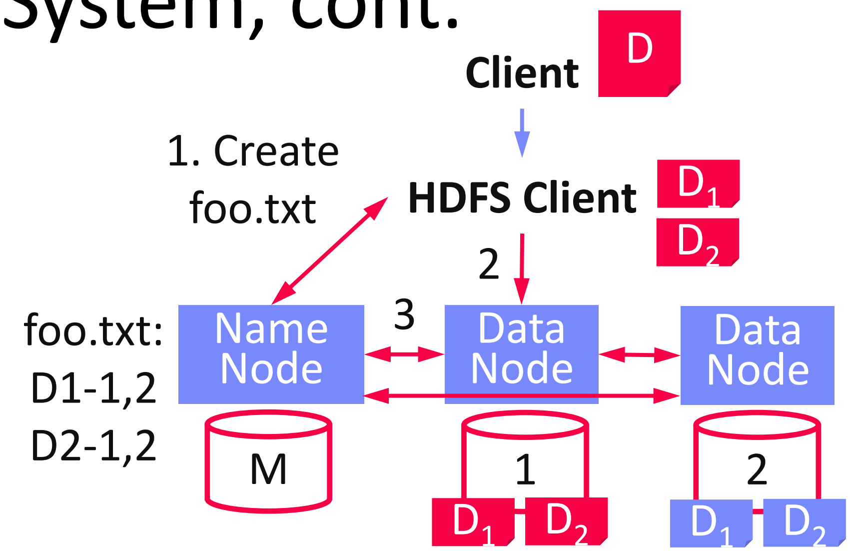


```
-rw-r--r--   3 mboehm hdfs  104510159 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00000
-rw-r--r--   3 mboehm hdfs  137887319 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00001
-rw-r--r--   3 mboehm hdfs  139012247 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00002
-rw-r--r--   3 mboehm hdfs  139123247 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00003
-rw-r--r--   3 mboehm hdfs  139053743 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00004
-rw-r--r--   3 mboehm hdfs  138928955 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00005
-rw-r--r--   3 mboehm hdfs  139016375 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00006
-rw-r--r--   3 mboehm hdfs  139047923 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00007
-rw-r--r--   3 mboehm hdfs  139042307 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00008
-rw-r--r--   3 mboehm hdfs  139068143 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00009
-rw-r--r--   3 mboehm hdfs  139029875 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00010
-rw-r--r--   3 mboehm hdfs  138901043 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00011
-rw-r--r--   3 mboehm hdfs  139042763 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00012
-rw-r--r--   3 mboehm hdfs  139030751 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00013
-rw-r--r--   3 mboehm hdfs  139172051 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00014
-rw-r--r--   3 mboehm hdfs  138962735 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00015
-rw-r--r--   3 mboehm hdfs  139079495 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00016
-rw-r--r--   3 mboehm hdfs   63417008 2018-10-20 22:59 /user/mboehm/data/mnist1m.bin/0-m-00017
```

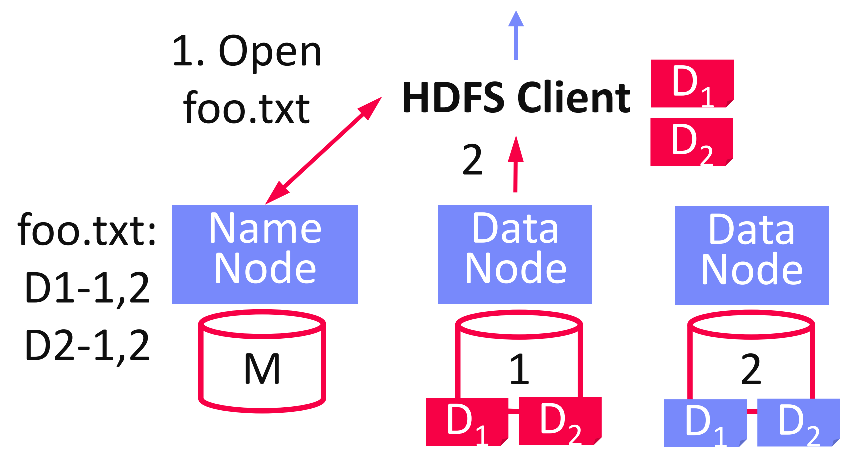
- **HDFS DataNode**
  - Worker daemon per cluster node that manages block storage (list of disks)
  - Block creation, deletion, replication as individual files in local FS
  - On startup: scan local blocks and send **block report** to name node
  - Serving block read and write requests
  - Send heartbeats to NameNode (capacity, current transfers) and receives replies (replication, removal of block replicas)

Hadoop Distributed File System, cont.

- **HDFS Write**
  - #1 Client RPC to NameNode to create file → lease/replica DNs
  - #2 Write blocks to DNs, pipelined replication to other DNs
  - #3 DNs report to NN via heartbeat



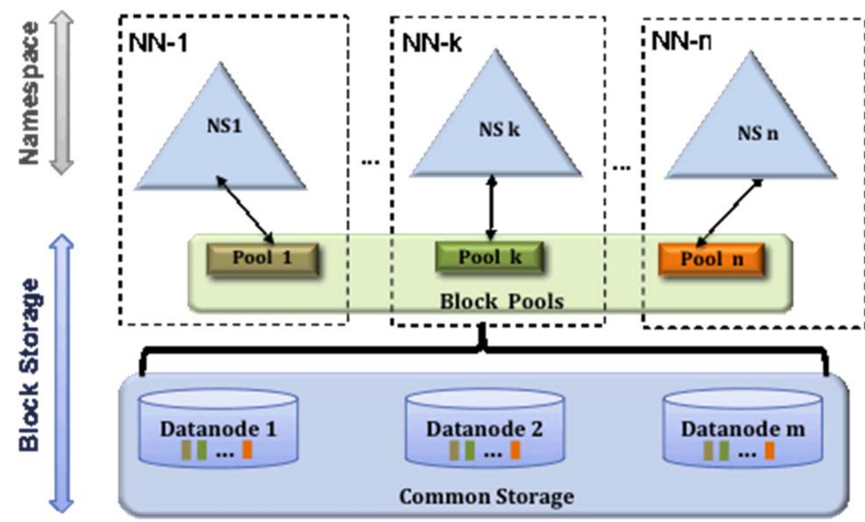
- **HDFS Read**
  - #1 Client RPC to NameNode to open file → DNs for blocks
  - #2 Read blocks sequentially from closest DN w/ block
  - InputFormats and RecordReaders as abstraction for multi-part files (incl. compression/encryption)



Hadoop Distributed File System, cont.

- **Data Locality**
  - HDFS is generally rack-aware (node-local, rack-local, other)
  - Schedule reads from closest data node
  - Replica placement (rep 3): local DN, other-rack DN, same-rack DN
  - MapReduce/Spark: locality-aware execution (function vs data shipping)

- **HDFS Federation**
  - Eliminate NameNode as namespace scalability bottleneck
  - Independent NameNodes, responsible for name spaces
  - DataNodes store blocks of all NameNodes
  - Client-side mount tables



[**Credit:** https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html]

Excursus: Amazon Redshift

[Anurag Gupta et al.: Amazon Redshift and the Case for Simpler Data Warehouses. **SIGMOD 2015**]

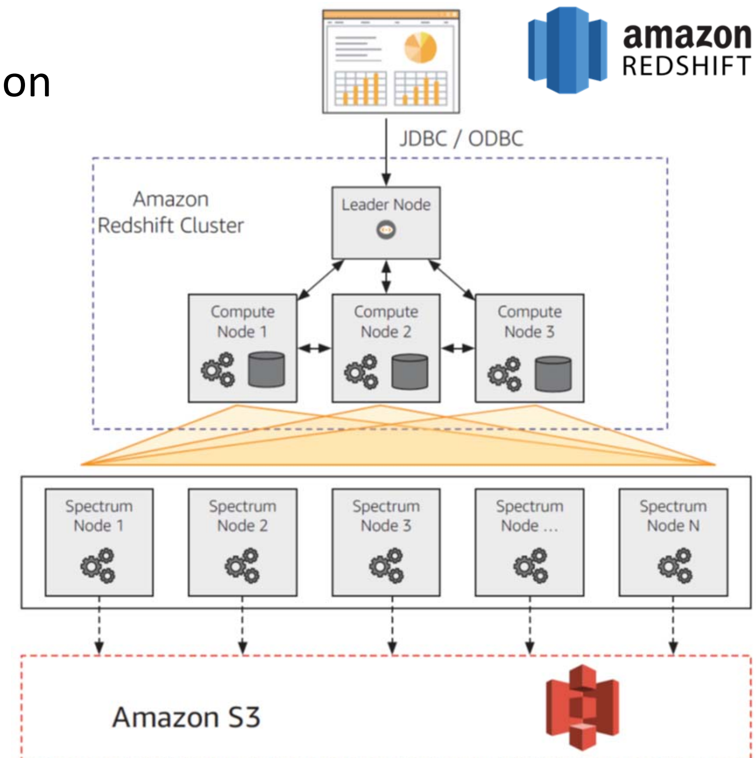
[Mengchu Cai et al.: Integrated Querying of SQL database data and S3 data in Amazon Redshift. **IEEE Data Eng. Bull. 41(2) 2018**]

- **Motivation** (release 02/2013)
  - **Simplicity and cost-effectiveness** (fully-managed DWH at petabyte scale)
- **System Architecture**
  - **Data plane:** data storage and **SQL** execution
  - **Control plane:** workflows for monitoring, and managing databases, AWS services
- **Data Plane**
  - Leader node + sliced compute nodes in **EC2** with **local storage**
  - Replication across nodes + **S3 backup**
  - **Query compilation** in C++ code
  - Support for **flat and nested files**
- **Similar Systems**

Google BigQuery

snowflake

Microsoft



Distributed Data Analysis

Data-Parallel Computation
(MapReduce, Spark)

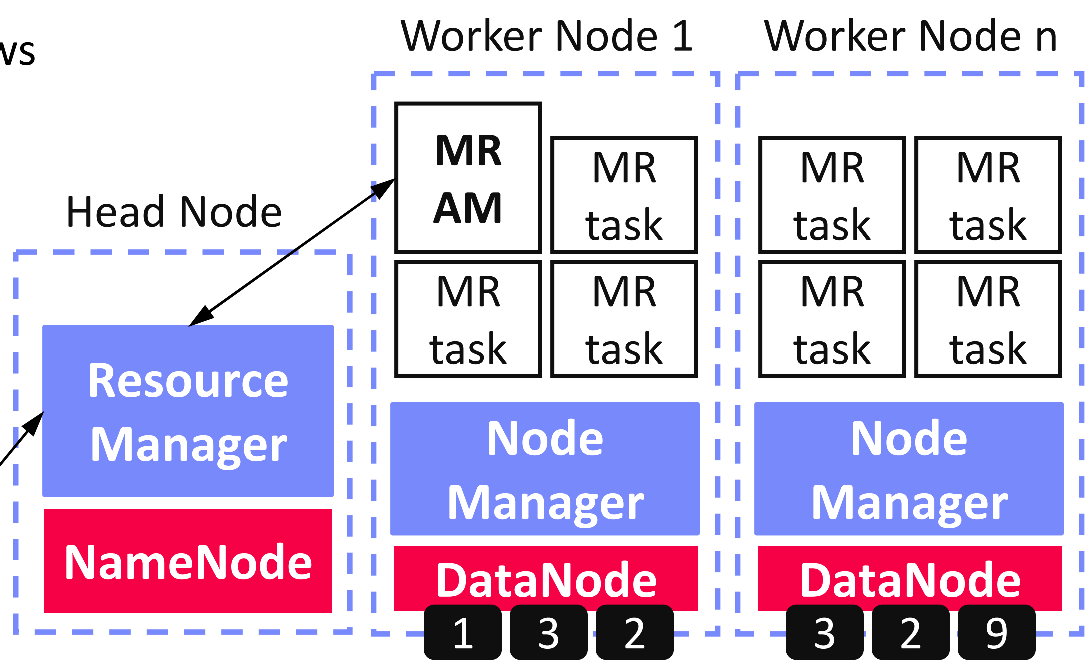
Hadoop History and Architecture

[Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. **OSDI 2004**]

- **Recap: Brief History**
  - Google's GFS [SOSP'03] + MapReduce
    → Apache Hadoop (2006)
  - Apache Hive (SQL), Pig (ETL), Mahout (ML), Giraph (Graph)

- **Hadoop Architecture / Eco System**
  - Management (Ambari)
  - Coordination / workflows (Zookeeper, Oozie)
  - Storage (**HDFS**)
  - Resources (**YARN**) [SoCC'13]
  - Processing (**MapReduce**)

MR Client



Central Data Abstractions

- **#1 Files and Objects**
  - **File:** Arbitrarily large sequential data in specific file format (CSV, binary, etc)
  - **Object:** binary large object, with certain meta data

- **#2 Distributed Collections**
  - Logical multi-set (**bag**) of **key-value pairs** (**unsorted collection**)
  - Different physical representations
  - **Easy distribution** of pairs via horizontal partitioning (aka shards, partitions)
  - Can be created from single file, or directory of files (unsorted)

Key	Value
4	Delta
2	Bravo
1	Alpha
3	Charlie
5	Echo
6	Foxtrott
7	Golf

MapReduce – Programming Model

- **Overview Programming Model**
  - Inspired by functional programming languages
  - **Implicit parallelism** (abstracts distributed storage and processing)
  - Map function: key/value pair → set of intermediate key/value pairs
  - Reduce function: merge all intermediate values by key

- **Example** `SELECT Dep, count(*) FROM csv_files GROUP BY Dep`

Name	Dep
X	CS
Y	CS
A	EE
Z	CS

Collection of key/value pairs

```
map(Long pos, String line) {
  parts ← line.split(“,”)
  emit(parts[1], 1)
}
```

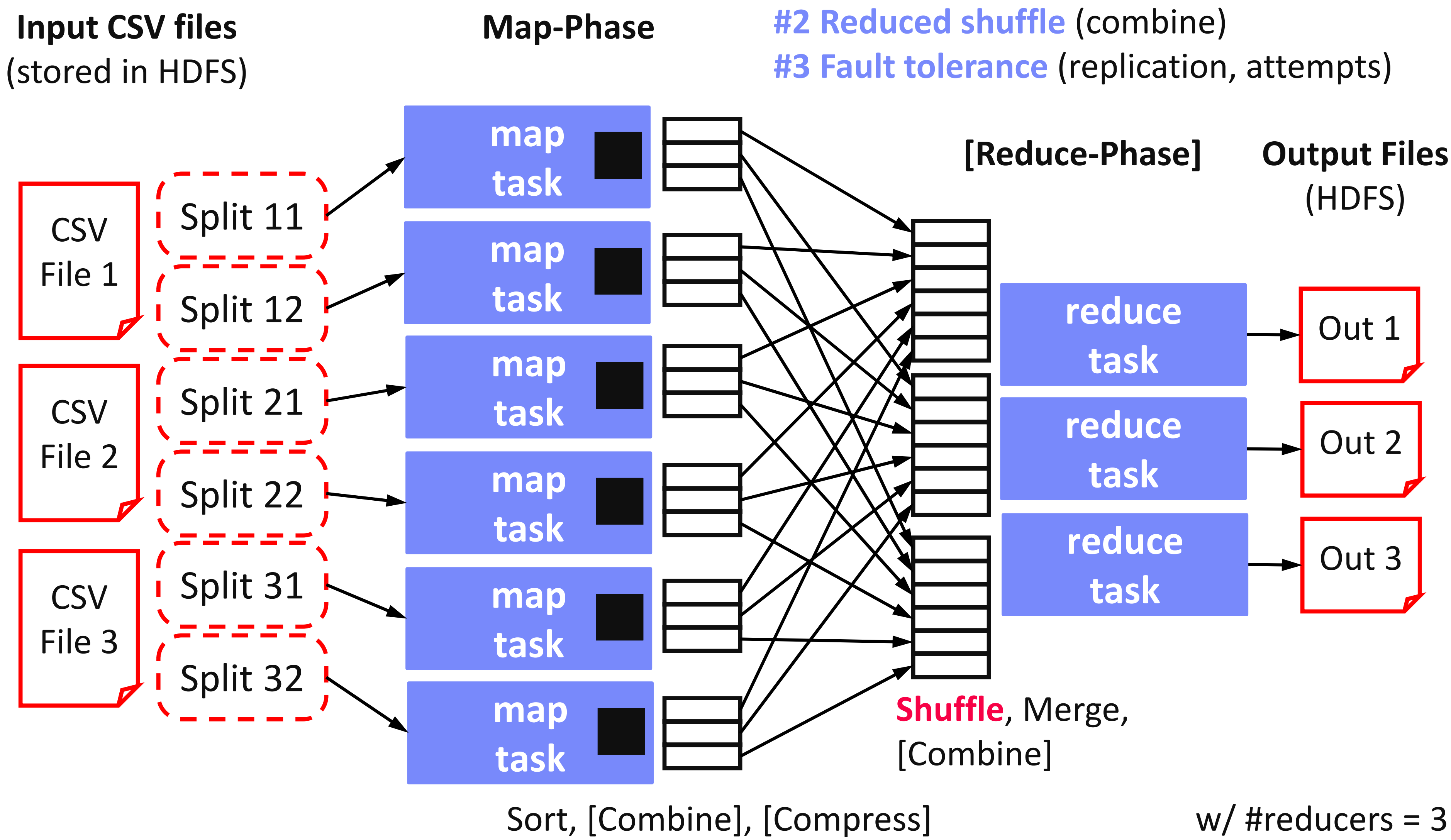
CS	1
CS	1
EE	1
CS	1

```
reduce(String dep,
       Iterator<Long> iter) {
  total ← iter.sum();
  emit(dep, total)
}
```

CS	3
EE	1

MapReduce – Execution Model

#1 Data Locality (delay sched., write affinity)
#2 Reduced shuffle (combine)
#3 Fault tolerance (replication, attempts)



Spark History and Architecture

- **Summary MapReduce**
  - Large-scale & fault-tolerant processing w/ UDFs and files → **Flexibility**
  - Restricted functional APIs → **Implicit parallelism and fault tolerance**
  - **Criticism**: #1 **Performance**, #2 **Low-level APIs**, #3 **Many different systems**

- **Evolution to Spark** (and Flink)
  - Spark [HotCloud'10] + RDDs [NSDI'12] → **Apache Spark** (2014)
  - **Design: standing executors with in-memory storage**, lazy evaluation, and fault-tolerance via RDD lineage
  - **Performance:** In-memory storage and fast job scheduling (100ms vs 10s)
  - **APIs:** Richer functional APIs and general computation DAGs, high-level APIs (e.g., DataFrame/Dataset), unified platform

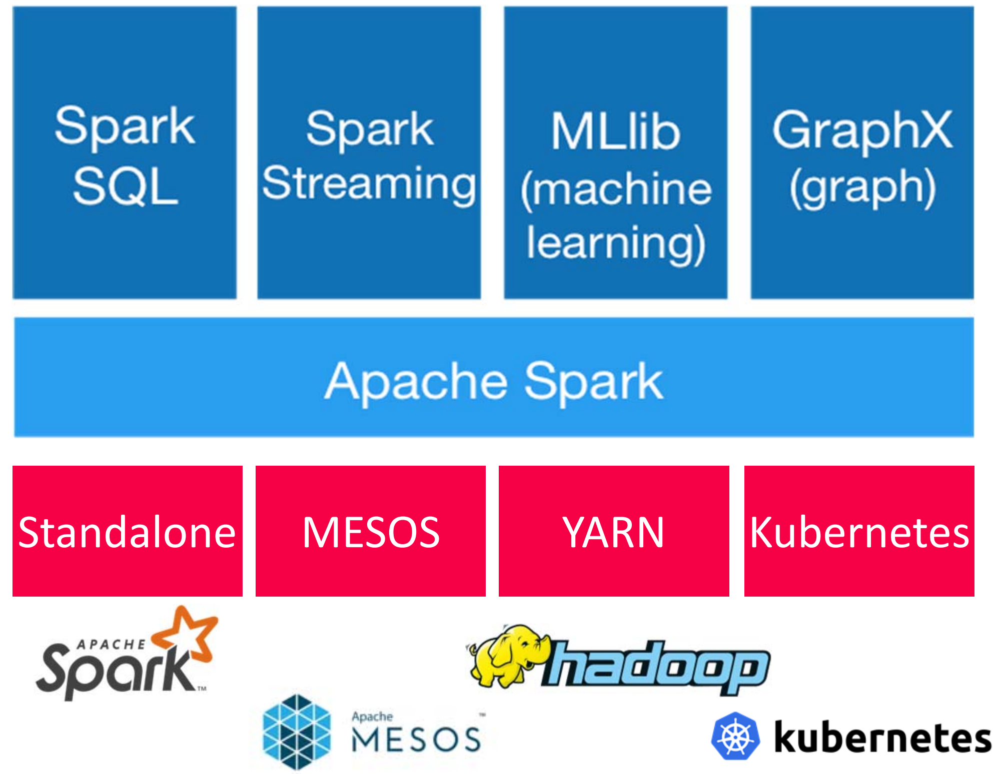
→ **But many shared concepts/infrastructure**

- **Implicit parallelism through dist. collections** (data access, fault tolerance)
- Resource negotiators (YARN, Mesos, Kubernetes)
- HDFS and object store connectors (e.g., Swift, S3)

Spark History and Architecture, cont.

- **High-Level Architecture**
  - **Different language bindings**: Scala, Java, Python, R
  - **Different libraries**: SQL, ML, Stream, Graph
  - Spark core (incl RDDs)
  - **Different cluster managers**: Standalone, Mesos, **Yarn**, **Kubernetes**
  - Different file systems/ formats, and data sources: **HDFS**, **S3**, SWIFT, **DBs**, **NoSQL**

[https://spark.apache.org/]



- **Focus on a unified platform for data-parallel computation**

Resilient Distributed Datasets (RDDs)

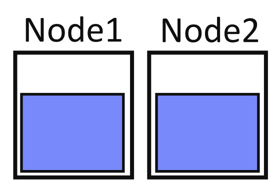
- **RDD Abstraction**
  - **Immutable**, partitioned **collections of key-value pairs**
  - **Coarse-grained** deterministic operations (transformations/actions)
  - Fault tolerance via lineage-based re-computation

```
JavaPairRDD
  <MatrixIndexes,MatrixBlock>
```

- **Operations**
  - Transformations: define new RDDs
  - Actions: return result to driver

Type	Examples
Transformation (**lazy**)	`map`, `hadoopFile`, `textFile`, `flatMap`, `filter`, `sample`, `join`, `groupByKey`, `cogroup`, `reduceByKey`, `cross`, `sortByKey`, `mapValues`
Action	`reduce`, `save`, `collect`, `count`, `lookupKey`

- **Distributed Caching**
  - Use fraction of worker **memory for caching**
  - Eviction at granularity of individual partitions
  - **Different storage levels** (e.g., mem/disk x serialization x compression)



Spark Resilient Distributed Datasets (RDDs), cont.

- **Lifecycle of an RDD**
  - Note: can't broadcast an RDD directly

```
X.filter(foo())
X.mapValues(foo())
X.reduceByKey(foo())
X.cache()
```

```
sc.parallelize(lst)
```

Local Data (value, collection)

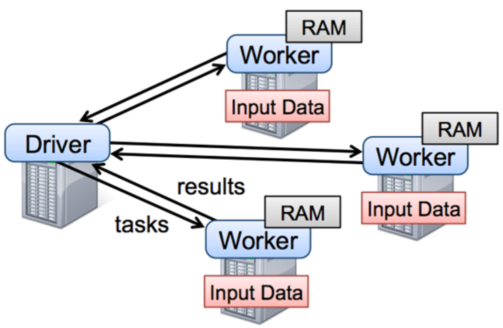
Distributed Collection

```
lst = X.collect()
v = X.reduce(foo())
```

```
sc.hadoopFile(f)
sc.textFile(f)
```

```
X.saveAsObjectFile(f)
X.saveAsTextFile(f)
```

File on DFS



Partitions and Implicit/Explicit Partitioning

- **Spark Partitions**
  - Logical key-value collections are split into **physical partitions**
  - Partitions are granularity of **tasks, I/O, shuffling, evictions**

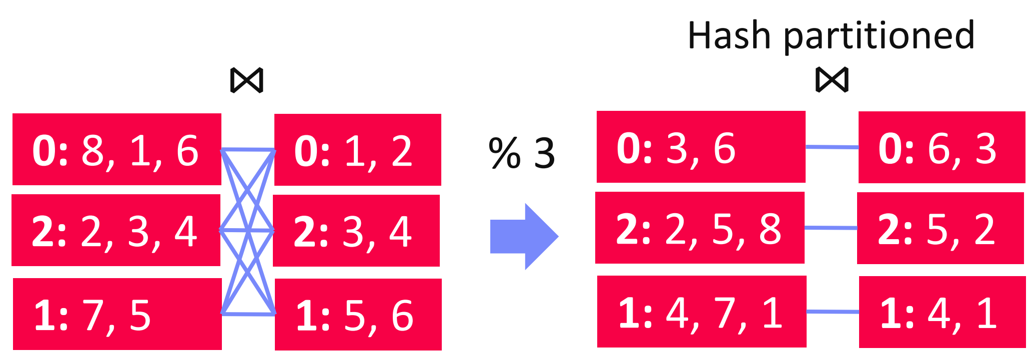
~128MB

- **Partitioning via Partitioners**
  - Implicitly on every data shuffling
  - Explicitly via `R.repartition(n)`

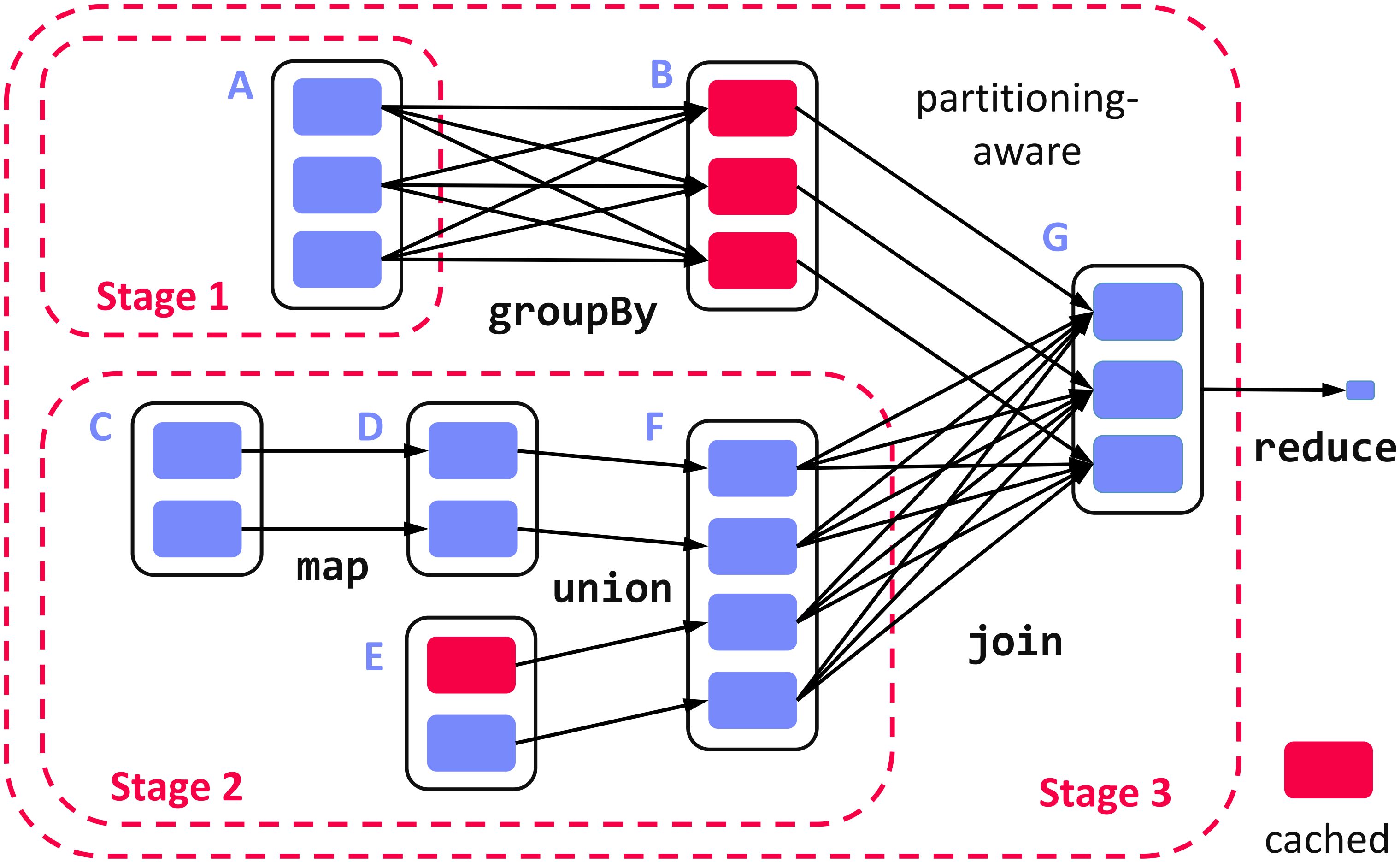
Example Hash Partitioning:
For all (k,v) of R:
pid = hash(k) % n

- **Partitioning-Preserving**
  - All operations that are guaranteed to keep keys unchanged (e.g. `mapValues()`, `mapPartitions()` w/ preservesPart flag)

- **Partitioning-Exploiting**
  - Join: R3 = R1.join(R2)
  - Lookups: `v = C.lookup(k)`



Spark Lazy Evaluation, Caching, and Lineage



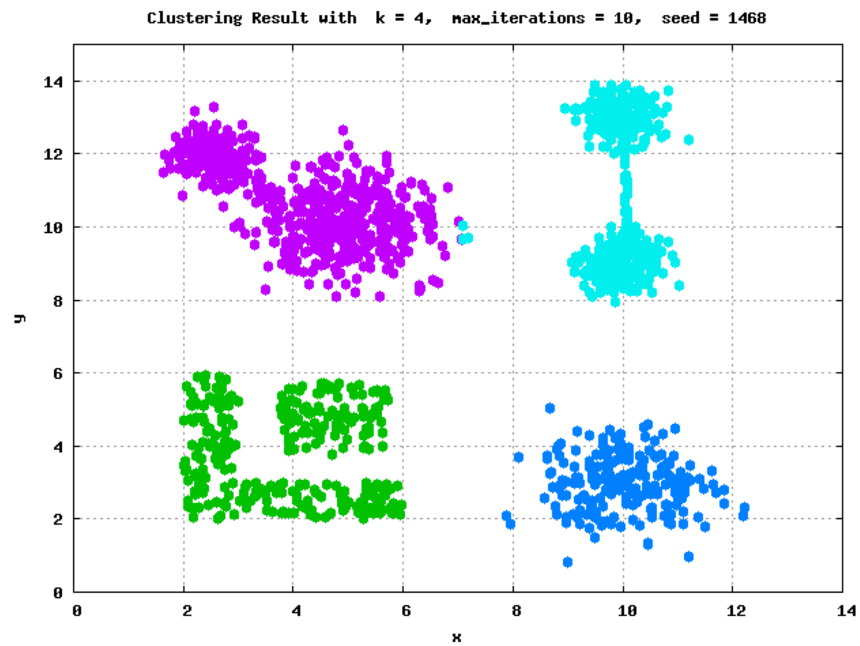
[Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, Ion Stoica: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. **NSDI 2012**]

Example: k-Means Clustering

- **k-Means Algorithm**
  - Given dataset D and number of clusters k, find cluster centroids ("mean" of assigned points) that minimize within-cluster variance
  - Euclidean distance: **sqrt**(**sum**((**a**-**b**)^2))

- **Pseudo Code**

```
function Kmeans(D, k, maxiter) {
  C' = randCentroids(D, k);
  C = {};
  i = 0; //until convergence
  while( C' != C & i<=maxiter ) {
    C = C';
    i = i + 1;
    A = getAssignments(D, C);
    C' = getCentroids(D, A, k);
  }
  return C'
}
```



Example: K-Means Clustering in Spark

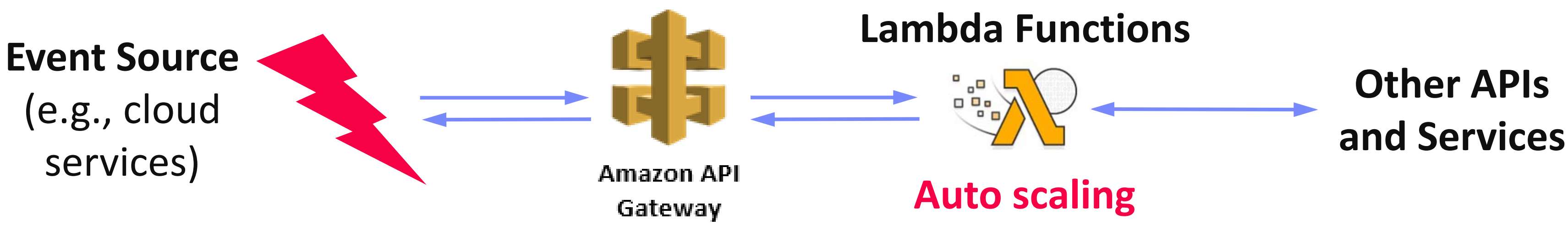
```
// create spark context (allocate configured executors)
JavaSparkContext sc = new JavaSparkContext();

// read and cache data, initialize centroids
JavaRDD<Row> D = sc.textFile("hdfs:/user/mboehm/data/D.csv")
  .map(new ParseRow()).cache(); // cache data in spark executors
Map<Integer,Mean> C = asCentroidMap(D.takeSample(false, k));

// until convergence
while( !equals(C, C2) & i<=maxiter ) {
  C2 = C; i++;
  // assign points to closest centroid, recompute centroid
  Broadcast<Map<Integer,Row>> bC = sc.broadcast(C)
  C = D.mapToPair(new NearestAssignment(bC))
       .foldByKey(new Mean(0), new IncComputeCentroids())
       .collectAsMap();
}

return C;
```

Note: Existing library algorithm
[https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/clustering/KMeans.scala]

Serverless Computing

[Joseph M. Hellerstein et al: Serverless Computing: One Step Forward, Two Steps Back. **CIDR 2019**]

- **Definition Serverless**
  - **FaaS:** functions-as-a-service (event-driven, stateless input-output mapping)
  - Infrastructure for deployment and auto-scaling of APIs/functions
  - Examples: **Amazon Lambda**, **Microsoft Azure Functions**, etc



Auto scaling
Pay-per-request
(1M x 100ms = 0.2$)

- **Example**

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class MyHandler implements RequestHandler<Tuple, MyResponse> {
    @Override
    public MyResponse handleRequest(Tuple input, Context context) {
       return expensiveStatelessComputation(input);
    }
}
```

Exercise 4:
Large-Scale Data Analysis

Published: May 24

Deadline: June 16

Task 4.1 Apache Spark Setup

3/25 points

- **#1 Pick your Spark Language Binding**
  - Java, Scala, Python
- **#2 Install Dependencies**
  - Java: Maven
    spark-core, spark-sql
  - Python:
    pip install pyspark

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-core_2.11</artifactId>
  <version>2.4.3</version>
 </dependency>
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.4.3</version>
</dependency>
```

- **(#3 Win Environment)**
  - Download https://github.com/steveloughran/winutils/tree/master/hadoop-2.7.1/bin/winutils.exe
  - Create environment variable HADOOP_HOME="<some-path>/hadoop"

Task 4.2 SQL Query Processing

4/25 points

- **Q12: Top 5 Co-Authors**
  - Compute top-5, unique co-author pairs by number of joint papers
  - Exclude duplicates (A1-A2, A2-A1)
  - Return names and paper count, sorted desc by #papers

	a1 character varying (128)	a2 character varying (128)	cnt bigint
1	Xuemin Lin	Wenjie Zhang 0001	83
2	Xuemin Lin	Ying Zhang 0001	70
3	Jianhua Feng	Guoliang Li 0001	67
4	Thomas Neumann 0001	Alfons Kemper	66
5	Yiqun Liu	Shaoping Ma	66

- **Q13: SIGMOD/PVLDB Papers**
  - Compute which persons published >20 SIGMOD/PVLDB papers between 2014 and 2020 (inclusive)
  - Return names and paper count, sorted desc by #papers

	name character varying (128)	count bigint
1	Lei Chen 0002	42
2	Guoliang Li 0001	40
3	Samuel Madden	36
4	Tim Kraska	32
5	Jeffrey Xu Yu	31
6	H. V. Jagadish	30
7	Xuemin Lin	30
8	Divesh Srivastava	30
9	Michael Stonebraker	27
10	Surajit Chaudhuri	27
11	Xiaokui Xiao	27
12	Aditya G. Parameswaran	26
13	Andrew Pavlo	26
14	Lu Qin	26
15	Gang Chen 0001	25
16	Beng Chin Ooi	25
17	Gautam Das 0001	25
18	Anastasia Ailamaki	24
19	Nan Tang 0001	24
20	Mourad Ouzzani	24
21	Stratos Idreos	23
22	Magdalena Balazinska	23
23	Fatma Özcan	23
24	Carsten Binnig	23
25	Kian-Lee Tan	22
26	Thomas Neumann 0001	22
27	Jignesh M. Patel	22
28	Donald Kossmann	21
29	Michael J. Franklin	21
30	Jian Pei	21
31	Bin Cui 0001	21
32	Divyakant Agrawal	21
33	Sihem Amer-Yahia	21

Task 4.3 Query Processing via Spark RDDs

12/25 points

- **#1 Spark Context Creation**
  - Create a spark context sc w/ local master (`local[*]`)
- **#2 Implement Q12 via RDD Operations**
  - Implement Q12 self-contained in `executeQ12RDD()`
  - All reads should use `sc.textFile(fname)`
  - RDD operations only → stdout
- **#3 Implement Q13 via RDD Operations**
  - Implement Q13 self-contained in `executeQ13RDD()`
  - All reads should use `sc.textFile(fname)`
  - RDD operations only → stdout

See Spark online documentation for details

Task 4.4 Query Processing via Spark SQL

6/25 points

- **#1 Spark Session Creation**
  - Create a spark session via a spark session builder and w/ local master (`local[*]`)
- **#2 Implement Q12 via Dataset Operations**
  - Implement Q12 self-contained in executeQ09Dataset()
  - All reads should use `sc.read().format("csv")`
  - SQL or Dataset operations only → Parquet
- **#3 Implement Q13 via Dataset Operations**
  - Implement Q13 self-contained in executeQ10Dataset()
  - All reads should use `sc.read().format("csv")`
  - SQL or Dataset operations only → Parquet

→ SQL processing of high importance in modern data management

See Spark online documentation for details

- **WebUI**

```
INFO Utils: Successfully started service 'SparkUI' on port 4040.
INFO SparkUI: Bound SparkUI to […] http://192.168.108.220:4040
```

Task 4.5 Extra Credit: Graph Processing

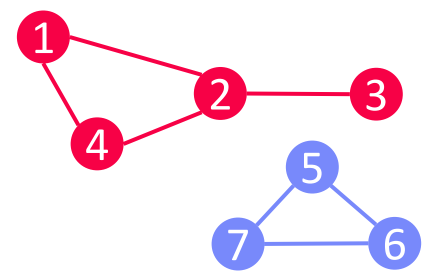
+5 points

- **Input Co-author graph**
  - AuthPapersCOO.csv (coordinate format)
    AuthPapersCSR.csv (compressed sparse row)

```
author,co-author
1001634,70215
1001634,519925
```

```
author,co-authors
1001634,70215:519925:1444319:2383440
1243968,76416:323847:407298:688292:918500:1198961:1231227:1256611:1377989
```

- **#1 Compute Connected Components**
  - Leverage Spark to compute assignment of vertices to components
  - Write output to text file, print #components to stdout
  - APIs up to you (e.g., Spark RDDs, Spark SQL, Spark GraphX)



  - Example Apache SystemDS

```
# initialize state with vertex ids
c = seq(1,nrow(G));
diff = Inf;
iter = 1;

# iterative computation of connected components
while( diff > 0 & (maxi==0 | iter<=maxi) ) {
  u = max(rowMaxs(G * t(c)), c);
  diff = sum(u != c)
  c = u; # update assignment
```

Conclusions and Q&A

- **Summary** **11 Distributed Storage & Data Analysis**
  - Cloud Computing Overview
  - Distributed Storage
  - Distributed Data Analytics

- **Next Lectures** (Part B: Modern Data Management)
  - **June 1:** Whit Monday (Pfingstmontag)
  - **12 Data stream processing systems** [Jun 08]