

# Data Integration and Large-scale Analysis (DIA)

## 09 Cloud Resource Management and Scheduling

**Prof. Dr. Matthias Boehm**

Technische Universität Berlin

Berlin Institute for the Foundations of Learning and Data

Big Data Engineering (DAMS Lab)

# Announcements / Administrative Items



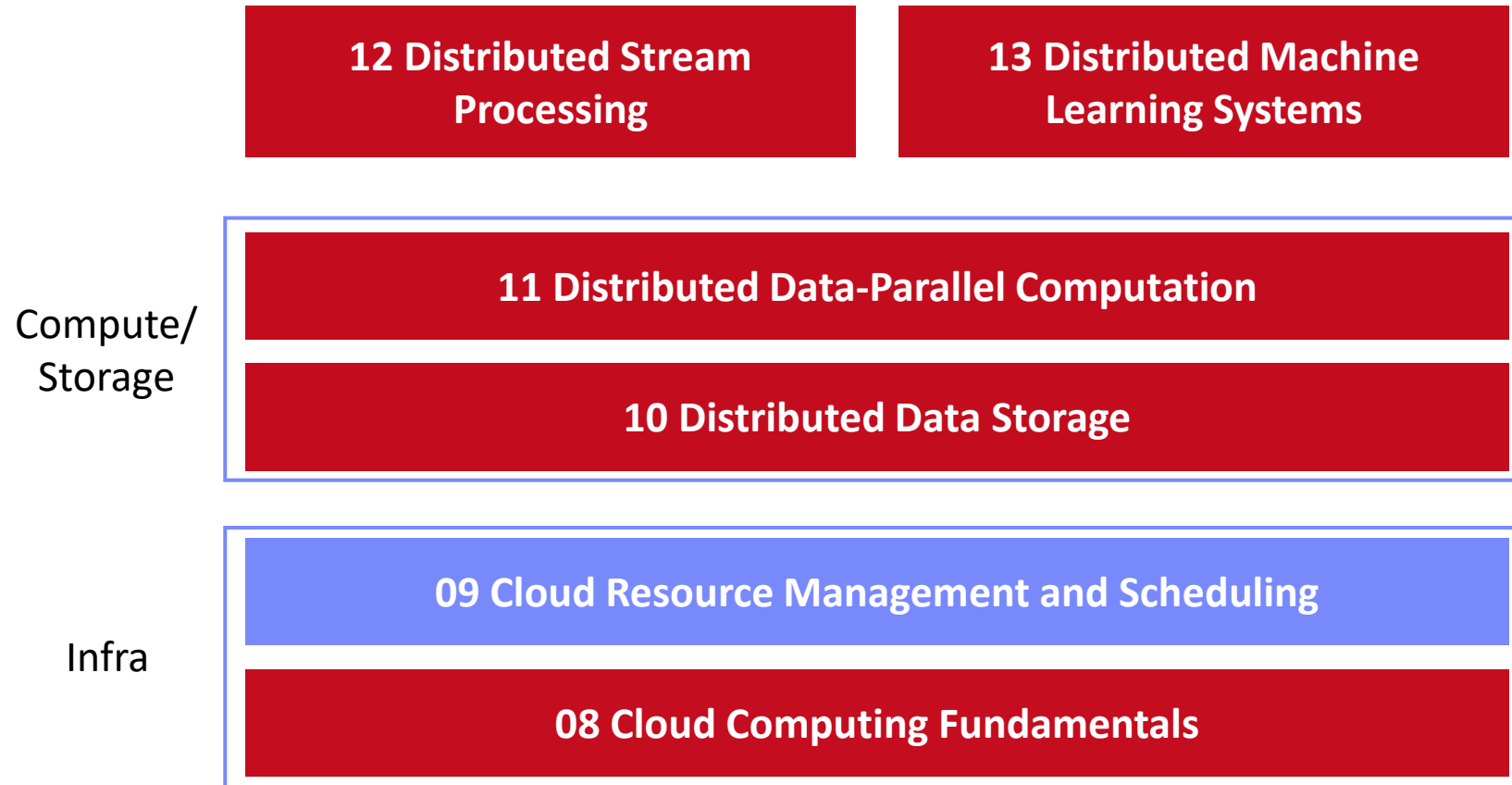
## ■ #1 Video Recording

- Hybrid lectures: in-person BH-N 243, zoom live streaming, video recording
- <https://tu-berlin.zoom.us/j/9529634787?pwd=R1ZsN1M3SC9BOU1OcFdmem9zT202UT09>

## ■ #2 Exercises/Projects

- **Reminder:** exercise/project submissions by **Jan 30** (no extensions)
- Make use of **virtual** / in-person (FR-766) office hours **Wed 5pm-6pm**

# Course Outline Part B: Large-Scale Data Management and Analysis



# Agenda



- **Motivation, Terminology, and Fundamentals**
- **Resource Allocation, Isolation, and Monitoring**
- **Task Scheduling and Elasticity**

# Motivation, Terminology, and Fundamentals

# Recap: 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

**“Computing as  
a Utility”**

## ■ 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

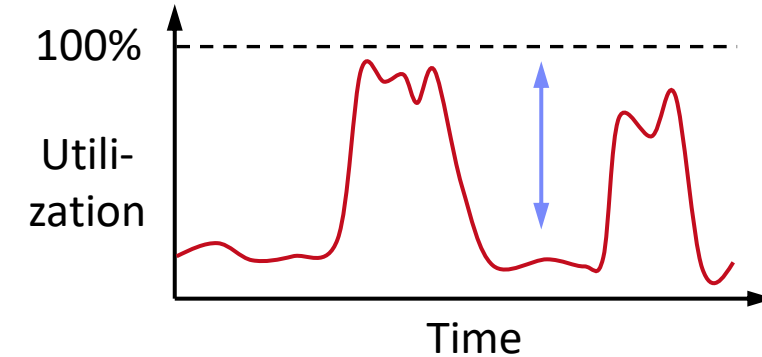
- Since ~2010 increasing move from on-prem to cloud resources
- System software licenses become increasingly irrelevant
- Few cloud providers dominate IaaS/PaaS/SaaS markets (w/ 2018 revenue):  
**Microsoft Azure Cloud** (\$ 32.2B), **Amazon AWS** (\$ 25.7B), **Google Cloud** (N/A), **IBM Cloud** (\$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

## Recap: 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

100 days @ 1 node

≈

1 day @ 100 nodes

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

### ■ Argument #3: Elasticity

- Assuming perfect scalability, work done in constant time \* resources
- Given virtually unlimited resources allows to reduce time as necessary

# Overview Resource Management & Scheduling

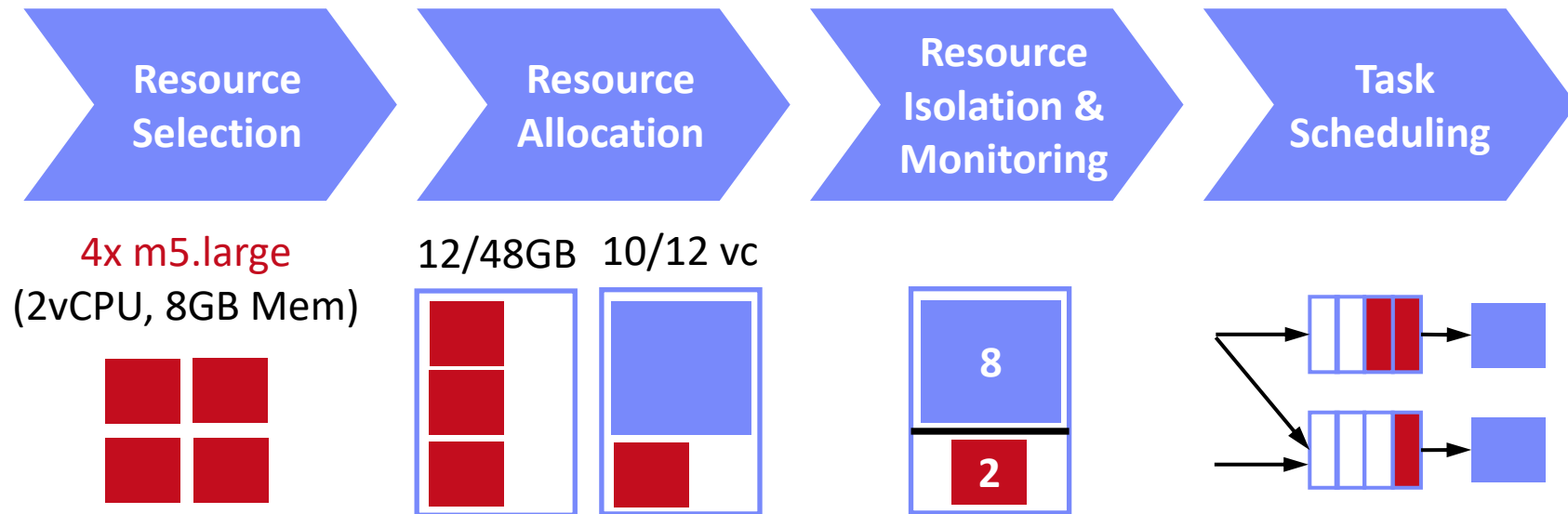
Scheduling is a fundamental computer science technique  
(at many different levels)



## ■ Resource Bundles

- Logical containers (aka nodes/instances) of different resources (**vc**ores, **mem**)
- Disk capacity, **disk** and **network** bandwidth
- Accelerator devices (**GPU**s, FPGAs), etc

## ■ Resource Management





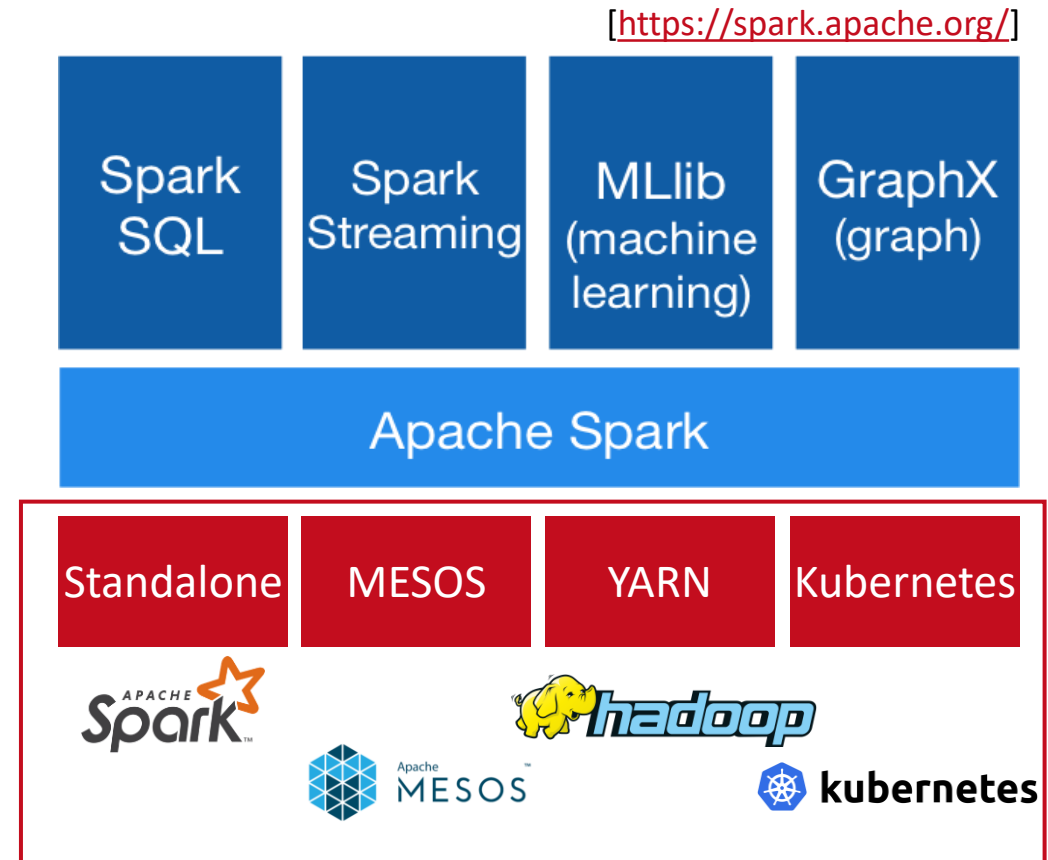
# Overview Resource Management & Scheduling, cont.



## ■ High-Level Architecture

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

➔ **Separation of concerns:**  
resource allocation vs task scheduling



# Scheduling Problems

[Eleni D. Karatza: Cloud Performance  
Resource Allocation and Scheduling Issues,  
Aristotle University of Thessaloniki 2018]

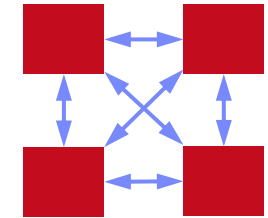


## ■ Bag-of-Tasks Scheduling

- Job of **independent** (embarrassingly parallel) tasks
- **Examples:** EC2 instances, map tasks

## ■ Gang Scheduling

- Job of frequently **communicating** parallel tasks
- **Examples:** MPI programs, parameter servers



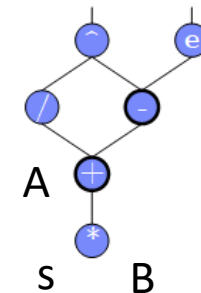
## ■ DAG Scheduling

- Job of tasks with **precedence constraints** (e.g., data dependencies)
- **Examples:** Op scheduling Spark, TensorFlow, SystemDS

$$C = A + s * B$$

$$D = (C/2)^{(C-1)}$$

$$E = \exp(C-1)$$



## ■ Real-Time Scheduling

- Job or task with associated deadline (soft/hard)
- **Examples:** rendering, car control



## ■ Operator-Device Placement

- Given neural network, multiple devices → operator placement (parallelism, data transfer)
- Sequence-to-sequence model to predict which operations should run on which device

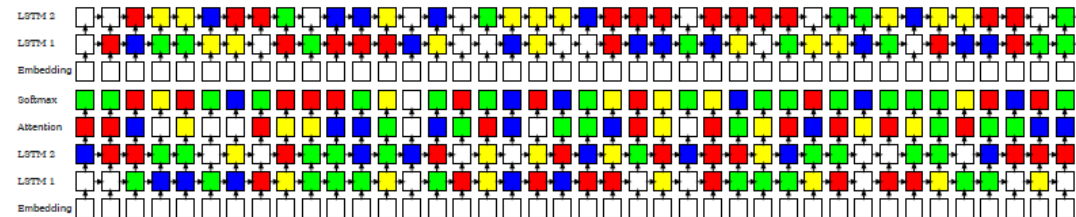
## ■ Example: ML Workloads

- white: CPU; colors: different GPU devices

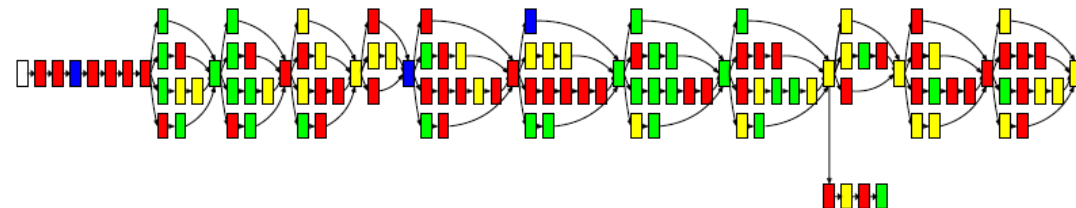
[Azalia Mirhoseini et al: Device Placement Optimization with Reinforcement Learning. **ICML 2017**]



Neural Machine  
Translation  
(RNN)



Inception V3  
(CNN)



# Basic Scheduling Metrics and Algorithms



## ■ Common Metrics

- **Mean time to completion** (total runtime for job), and max-stretch (completion/work – relative slowdown)
- **Mean response time** (job waiting time for resources);  
**Throughput** (jobs per time unit)
- **Constraints / SLOs**: max monetary costs, max latency, deadline

Service Level Agreements (**SLA**)  
→ Service Level Objectives (**SLO**)  
→ Service Level Indicators (**SLI**)

## ■ #1 **FIFO (first-in, first-out)**

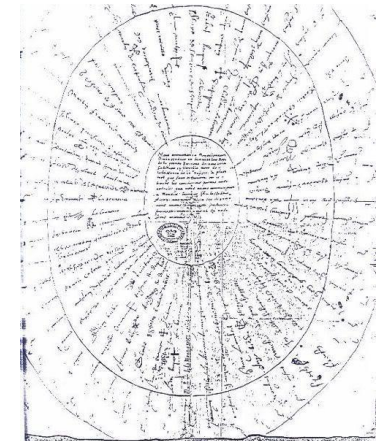
- Simple queueing and processing in order
- **Problem**: Single long-running job can stall many short jobs

## ■ #2 **SJF (shortest job first)**

- Sort jobs by expected runtime and execute in order ascending
- **Problem**: Starvation of long-running jobs

## ■ #3 **Round-Robin (FAIR)**

- Allocate similar time (tasks, time slices) to all jobs



[Credit:  
<https://en.wikipedia.org>  
(French "ruban rond" –  
English round ribbon)]

# Resource Allocation, Isolation, and Monitoring



## ■ #1 Manual Selection

- Rule of thumb (I/O, mem, CPU characteristics of app)
- Data characteristics, and framework configurations, experience

## ■ Example Spark Submit

```
export HADOOP_CONF_DIR=/etc/hadoop/conf
SPARK_HOME=../spark-2.4.0-bin-hadoop2.7
```

```
$SPARK_HOME/bin/spark-submit \
  --master yarn --deploy-mode client \
  --driver-java-options "-server -Xms40g -Xmn4g" \
  --driver-memory 40g \
  --num-executors 10 \
  --executor-memory 100g \
  --executor-cores 32 \
  SystemDS.jar -f test.dml -stats -explain -args ...
```

# Resource Selection, cont.



## ■ #2 Application-Agnostic, Reactive

- Dynamic allocation based on workload characteristics
- **Examples:** Spark dynamic allocation, Databricks AutoScaling

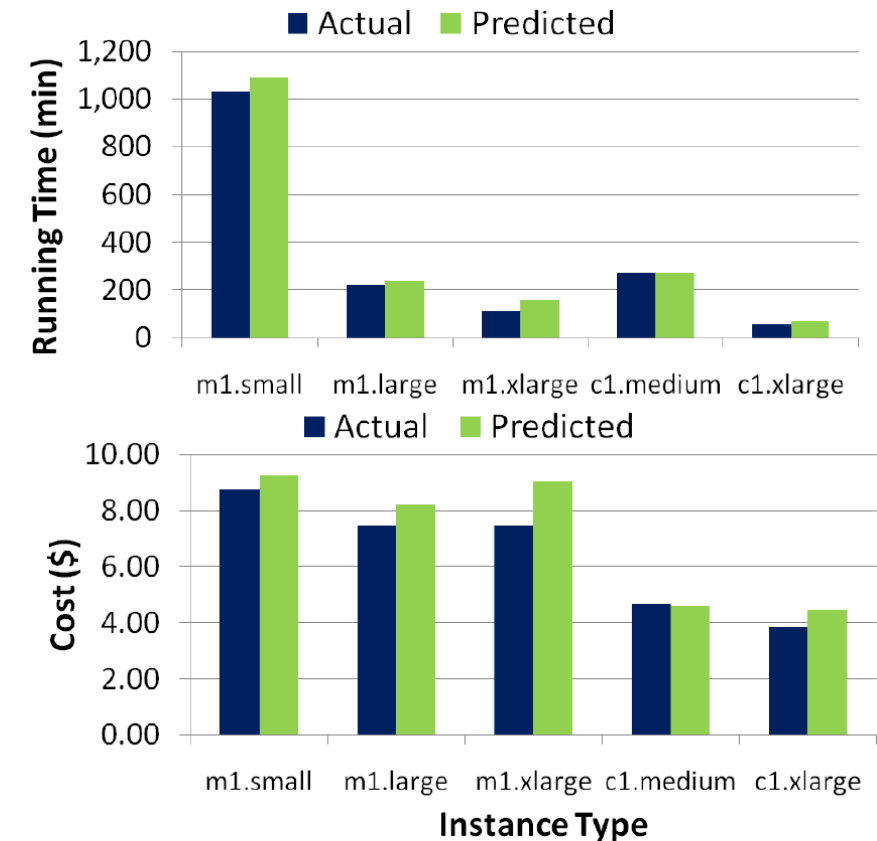
## ■ #3 Application-Aware, Proactive

- Estimate time/costs of job under different configurations (what-if scenario analysis)
- Min \$costs under time constraint
- Min runtime under \$cost constraint



[Herodotos Herodotou, Fei Dong, Shivnath Babu:  
No one (cluster) size fits all: automatic cluster  
sizing for data-intensive analytics. **SoCC 2011**]

(fixed MR job w/ 6 nodes)



# Resource Negotiation and Allocation

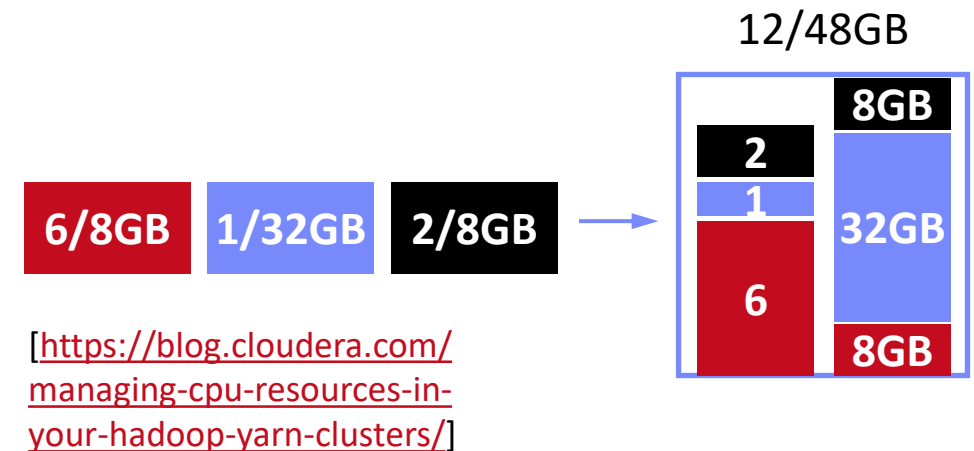
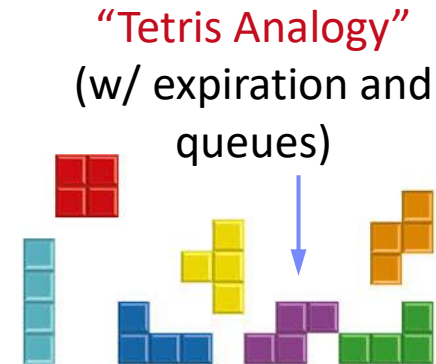


## ■ Problem Formulation

- N nodes with memory and CPU constraints
  - Stream of jobs with memory and CPU requirements
  - Assign jobs to nodes (or to minimal number of nodes)
- **Knapsack problem** (**bin packing problem**)

## ■ In Practice: Heuristics

- Major concern: **scheduling efficiency** (online, cluster bottleneck)
- Approach: **Sample queues**, **best/next-fit** selection
- Multiple metrics: **dominant resource calculator**





## ■ Slurm Overview

- Simple Linux Utility for Resource Management (SLURM)
- Heavily used in **HPC clusters** (e.g., MPI gang scheduling)



## ■ Scheduler Design

- Allocation/placement of requested resources
- Considers nodes, sockets, cores, HW threads, memory, GPUs, file systems, SW licenses
- **Job submit options:**  
**sbatch** (async job script), **salloc** (interactive); **srun** (sync job submission and scheduling)
- **Configuration:** cluster, node count (ranges), task count, mem, etc
- **Constraints via filters:** sockets-per-node, cores-per-socket, threads-per-core  
mem, mem-per-cpu, mincpus, tmp min-disk-space
- Elasticity via re-queueing

[Don Lipari: The SLURM Scheduler Design, User Group Meeting, **2012**]

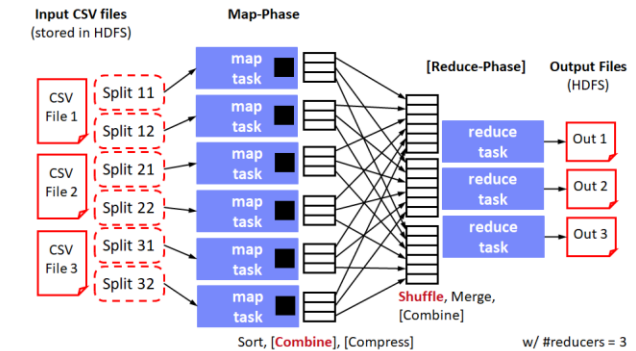


# Background: Hadoop JobTracker (anno 2012)



## ■ Overview

- Hadoop cluster w/ fixed configuration of **n map** slots, **m reduce slots** (fixed number and fixed memory config map/reduce tasks)
- JobTracker schedules map and reduce tasks to slots
- FIFO and FAIR schedulers, account for data locality



## ■ Data Locality

- Levels: **data local**, **rack local**, **different rack**
- **Delay scheduling** (with FAIR scheduler)  
wait 1-3s for data local slot

[Matei Zaharia et al: Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. **EuroSys 2010**]



## ■ Problem

- Intermixes resource allocation and task scheduling → **Scalability problems in large clusters**
- Forces every application into MapReduce programming model

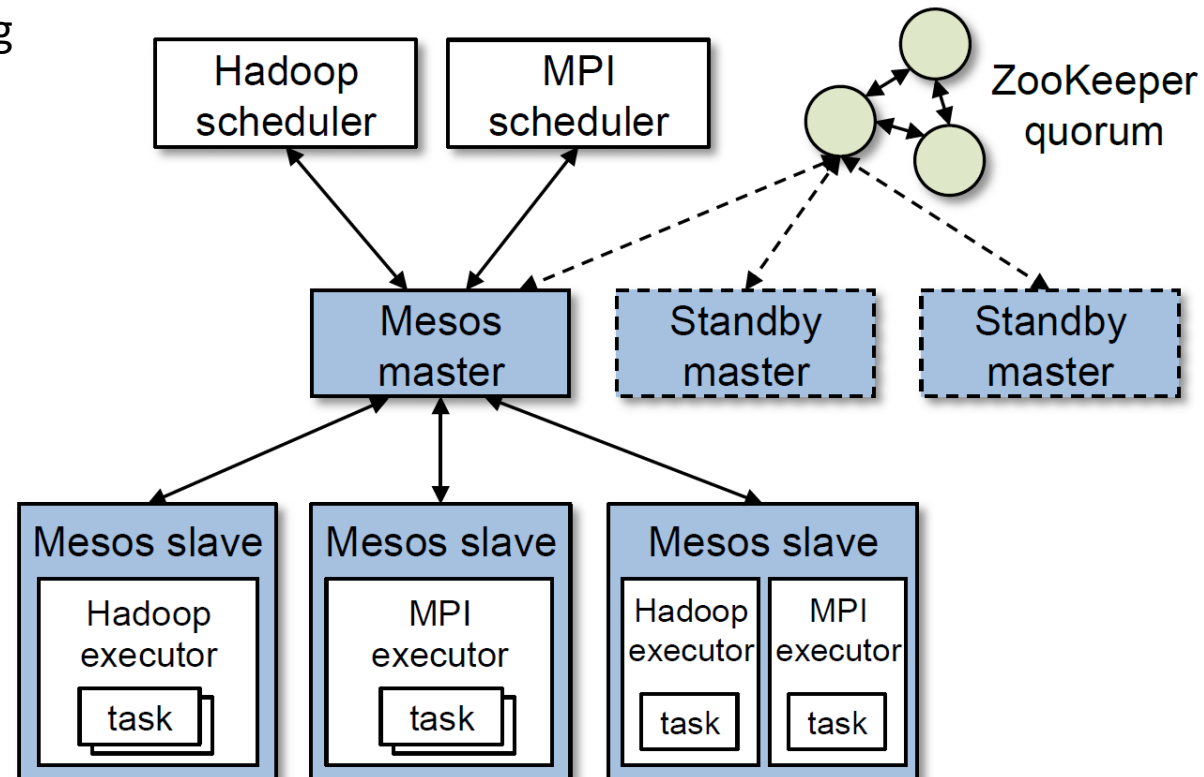
# Mesos Resource Management

[Benjamin Hindman et al: Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. **NSDI 2011**]



## ■ Overview Mesos

- Fine-grained, **multi-framework cluster sharing**
- Scalable and efficient scheduling  
→ **delegated to frameworks**
- **Resource offers**



# Mesos Resource Management, cont.

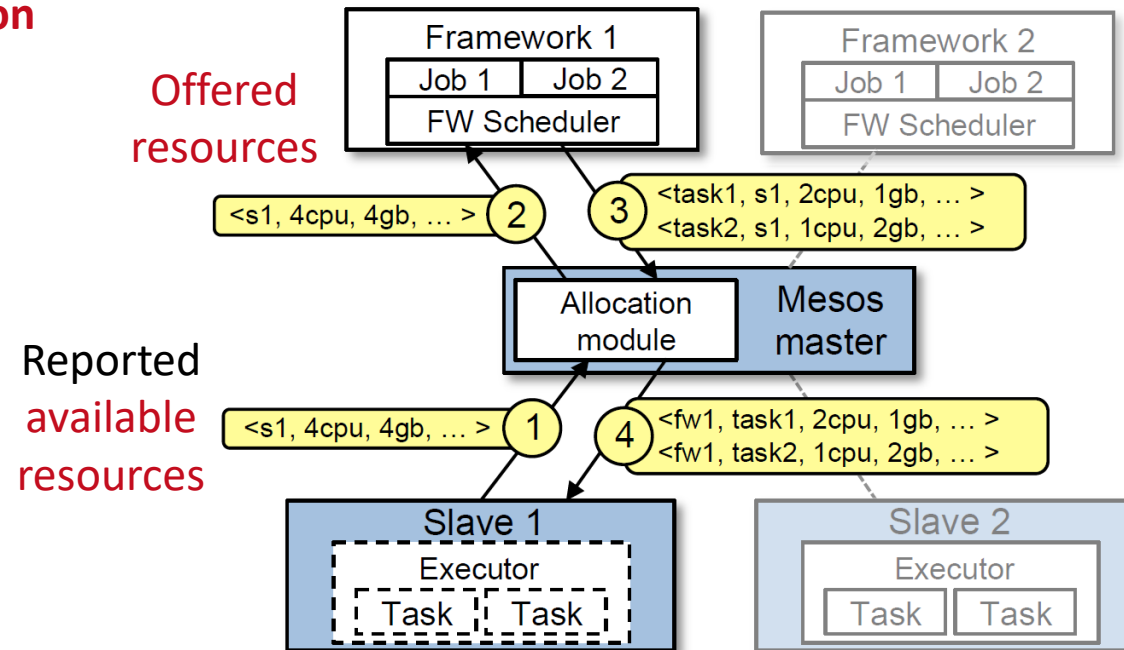
[Benjamin Hindman et al: Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. **NSDI 2011**]



## Resource Offers

- Mesos master decides how many resources to offer
- Framework scheduler decides which offered resources to accept/reject
- **Challenge:** long waiting times, lots of offers

→ **filter specification**



**Mesosphere  
Marathon:**  
container  
orchestration  
(e.g., Docker)

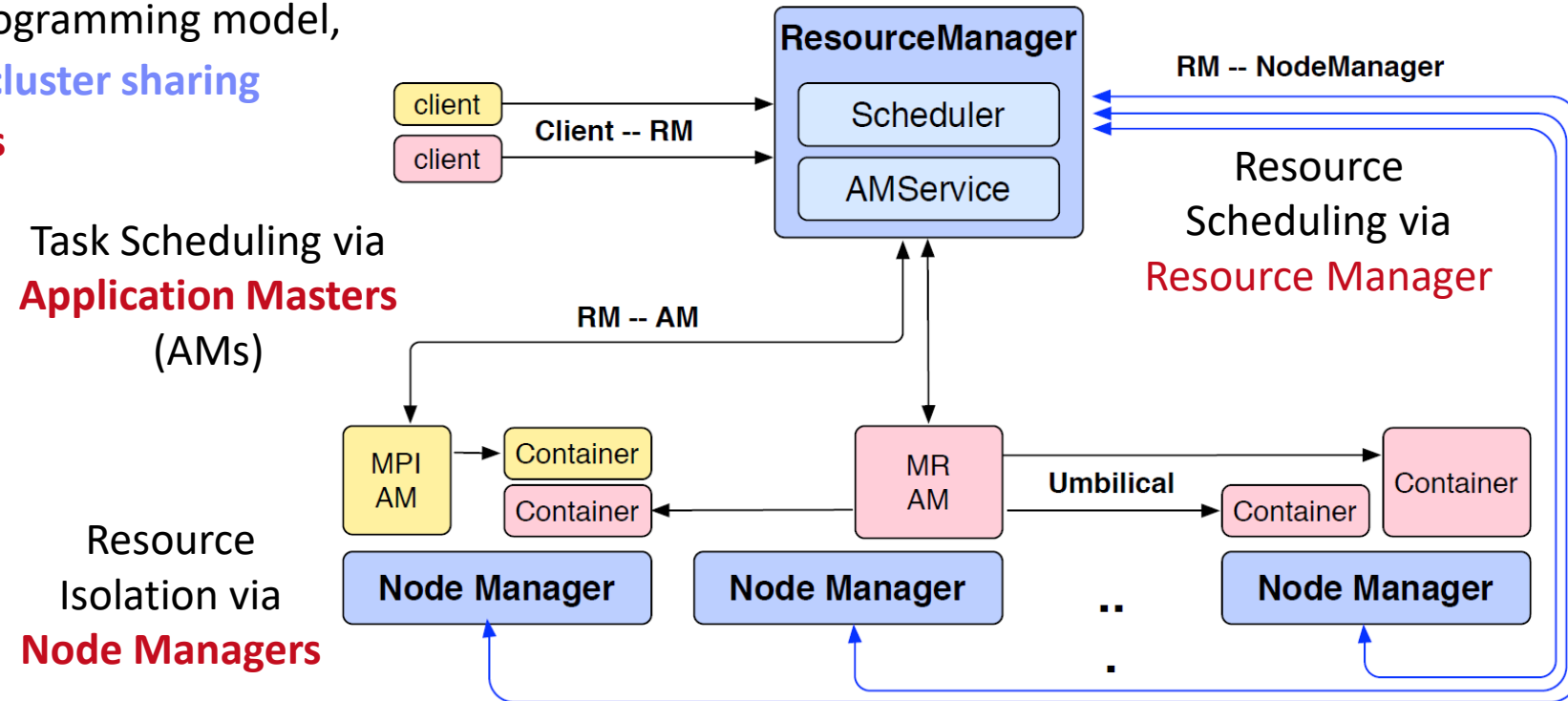
# YARN Resource Management

[Vinod Kumar Vavilapalli et al:  
Apache Hadoop YARN: yet another  
resource negotiator. **SoCC 2013**]



## ■ Overview YARN

- Hadoop 2 decoupled resource scheduler (negotiator)
- Independent of programming model,  
**multi-framework cluster sharing**
- **Resource Requests**



- **Example Apache SystemML AM Submission**  
(anno 2014)

```
// Set up the container launch context for the application master
ContainerLaunchContext amContainer =
    Records.newRecord(ContainerLaunchContext.class);
amContainer.setCommands(Collections.singletonList(command));
amContainer.setLocalResources(constructLocalResourceMap(yconf));
amContainer.setEnvironment(constructEnvironmentMap(yconf));

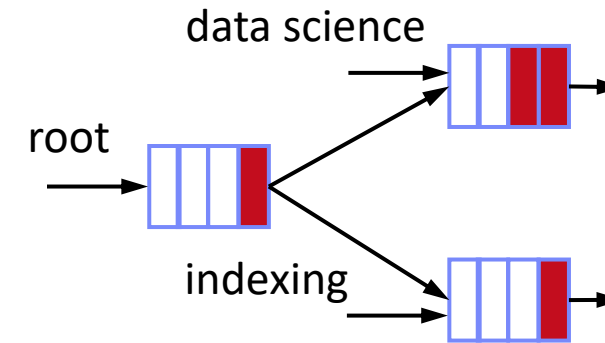
// Set up resource type requirements for ApplicationMaster
Resource capability = Records.newRecord(Resource.class);
capability.setMemory((int)computeMemoryAllocation(memHeap));
capability.setVirtualCores(numCores);

// Finally, set-up ApplicationSubmissionContext for the application
String qname = _dmlConfig.getTextValue(DMLConfig.YARN_APPQUEUE);
appContext.setApplicationName(APPMASTER_NAME); // application name
appContext.setAMContainerSpec(amContainer);
appContext.setResource(capability);
appContext.setQueue(qname); // queue (w/ min/max capacity constraints)

// Submit application (non-blocking)
yarnClient.submitApplication(appContext);
```

## ■ Capacity Scheduler

- **Hierarchy of queues** w/ shared resource among sub queues
- Soft (and optional hard) **[min, max]** constraints of max resources
- Default queue-user mapping
- No preemption during runtime (only redistribution over queues)



## ■ Fair Scheduler

- All applications get same resources over time
- Fairness decisions on memory requirements, but dominant resource fairness possible too

# Hydra: Federated RM @ Microsoft



## ■ Overview Hydra

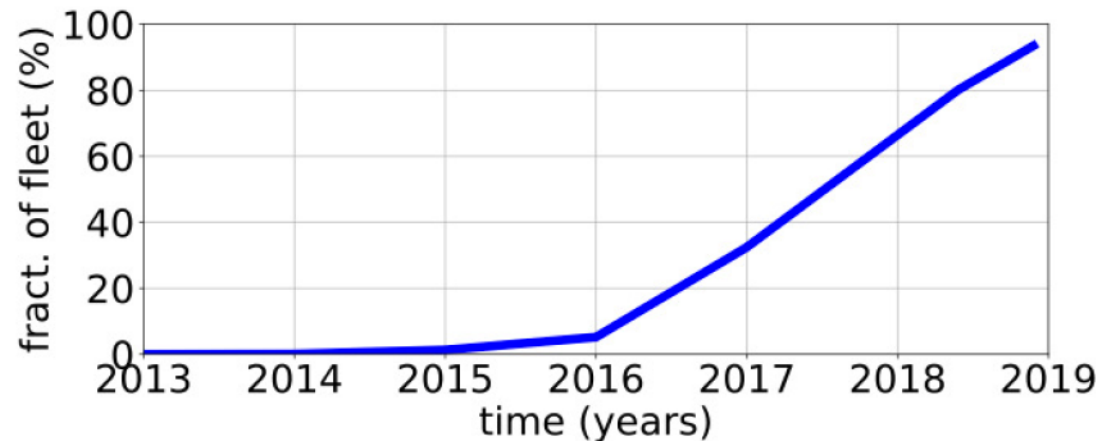
- Federated RM for internal MS big-data cluster
- Leverage **sub-clusters w/ YARN RM + router**
- AM-RM proxy (communication across sub clusters)
- Global policy generator + state store for runtime adaptation

[Carlo Curino et al.: Hydra: a federated resource manager for data-center scale analytics. **NSDI 2019**]



[[https://www.youtube.com/watch?v=kX13YamZXY&feature=emb\\_logo](https://www.youtube.com/watch?v=kX13YamZXY&feature=emb_logo)]

## ■ Deployment Statistics



**>250K** servers  
**>500K** daily jobs  
**>1 ZB** data processed  
**>1T** tasks scheduled  
(~2G tasks daily)  
**>70K** QPS (scheduling)  
**~60%** avg CPU util



# Kubernetes Container Orchestration



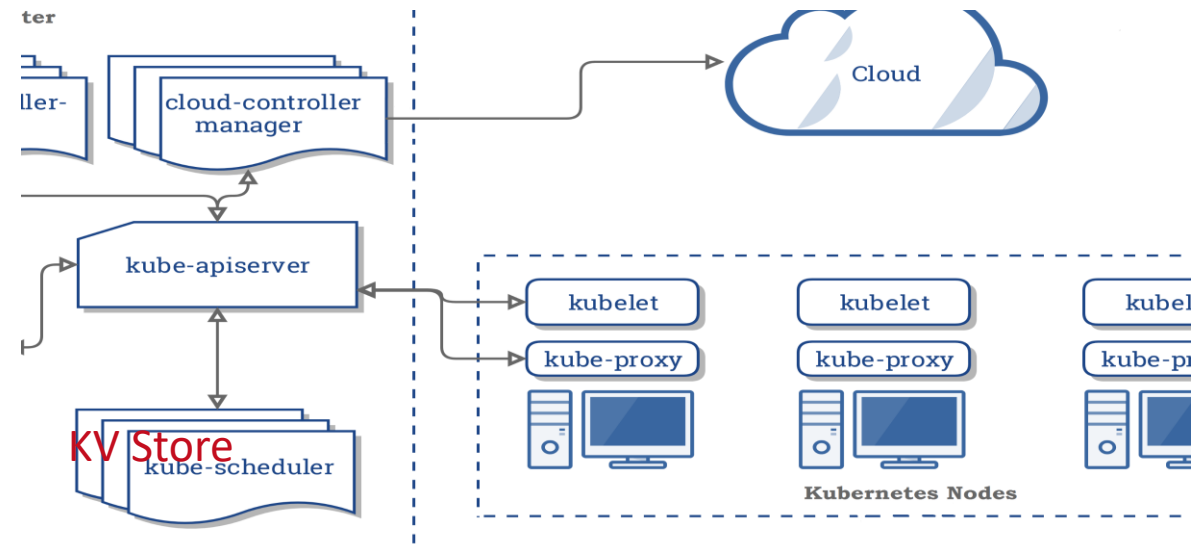
## ■ Overview Kubernetes

- **Open-source** system for automating, deployment, and management of containerized applications
- Container: resource isolation and application image

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

## ■ System Architecture

- **Pod**: 1 or more containers w/ individual IP
- **Kubelet**: node manager
- **Controller**: app master
- **API Server + Scheduler**
- Namespaces, quotas, access control, auth., logging & monitoring
- Wide variety of applications



[<https://kubernetes.io/docs/concepts/overview/components/>]



## ■ Pod Scheduling (Placement)

- Default scheduler: **kube-scheduler**, custom schedulers possible
  - **#1 Filtering**: finding feasible nodes for pod  
(resources, free ports, node selector, requested volumes, mem/disk pressure)
  - **#2 Scoring**: score feasible nodes → select highest score  
(spread priority, inter-pod affinity, requested priority, image locality)
  - Tuning: # scored nodes:  $\max(50, \text{percentageOfNodesToScore } [1, 100])$   
(sample taken round robin across zones)
- **Binding**: scheduler notifies API server

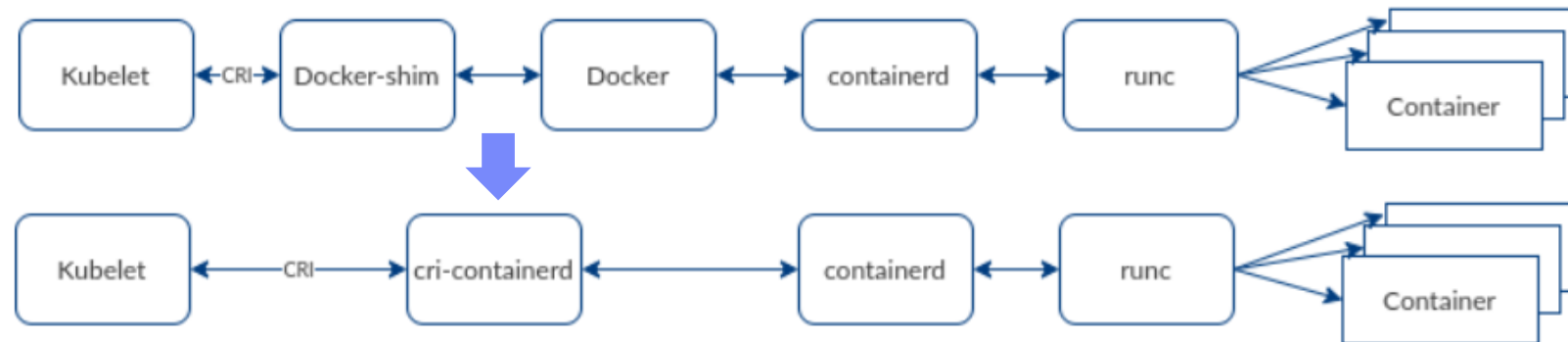
## ■ Container Stack

- Docker as stack of development and runtime services
- **containerd**: high-level daemon for image management
- **runc**: low-level container runtime

[<https://www.inovex.de/blog/containers-docker-containerd-nabla-kata-firecracker/>]

[Credit:

[www.inovex.de](https://www.inovex.de)]



## ■ Kubernetes deprecated Docker (as of 12/2020)

- Container Runtime Interface (CRI)
- Integrate other runtimes: cri-containerd, cri-o (Open Container Initiative)

[<https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/>]

## ■ Overview Key Primitives

- Platform-dependent resource isolation primitives → container runtime
  - **Linux namespaces:** restricting visibility
  - **Linux cgroups:** restricting usage
- } **Linux Containers**  
(e.g., basis of Docker)

## ■ Cgroups (Control Groups)

- Developed by Google engineers → Kernel 2.6.24 (2008)
- **Resource metering and limiting** (memory, CPU, block I/O, network)
- Each subsystem has a hierarchy (tree) with each node = group of processes
- Soft and hard limits on groups
  - **Mem** hard limit → triggers OOM killer (physical, kernel, total)
  - **CPU** → set weights (time slices)/no limits, cpuset to pin groups to CPUs

[Jérôme Petazzoni: Cgroups, namespaces and beyond: What are containers made from? DockerConEU 2015.]



[<https://www.youtube.com/watch?v=sK5i-N34im8&feature=youtu.be>]

# Resource Isolation, cont.

[<https://developer.ibm.com/hadoop/2017/06/30/deep-dive-yarn-cgroups/>]



## ■ Example YARN

- Set max CPU time per node manager
- Container weights: cores/total cores
- OOM killer if mem w/ overhead exceeded

```
<property>
  <name>yarn.nodemanager.resource.
    percentage-physical-cpu-limit<name>
  <value>60</value>
</property>      (hard → strict/soft)
```

## ■ Lesson Learned

- “The **resource isolation provided by containers has enabled Google to drive utilization significantly higher than industry norms.** [...] Borg uses containers to co-locate batch jobs with latency-sensitive, user-facing jobs on the same physical machines.”
- “The isolation is not perfect, though: **containers cannot prevent interference in resources that the operating-system kernel doesn’t manage,** such as level 3 processor caches and memory bandwidth [...]”

[Abhishek Verma et al. Large-scale cluster management at Google with Borg. **EuroSys 2015**]



[Malte Schwarzkopf et al.: Omega: flexible, scalable schedulers for large compute clusters. **EuroSys 2013**]



[Brendan Burns et al.: Borg, Omega, and Kubernetes. **ACM Queue 14(1): 10 (2016)**]



# Task Scheduling and Elasticity



# Task Scheduling Overview



## ■ Problem Formulation

- Given computation **job** and **set of resources** (servers, threads)
- Distribute job in pieces across resources

## ■ #1 Job-Task Partitioning

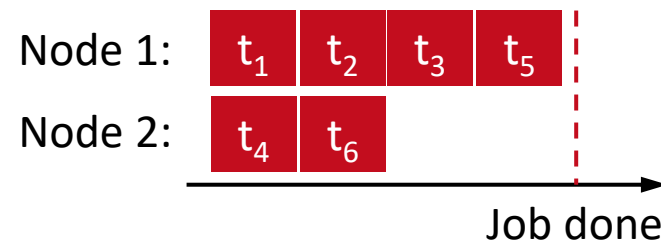
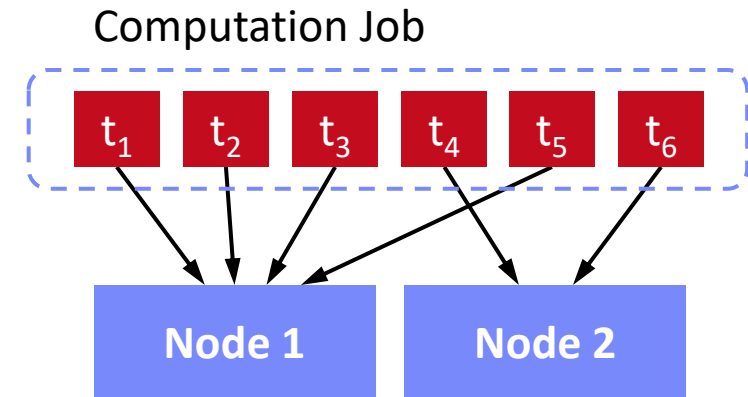
- Split job into sequence of N tasks

## ■ #2 Task Placement / Execution

- Assign tasks to K resources for execution

## ■ Goal: Min Job Completion Time

- **Beware:** Max runtime per resource determines job completion time



# Task Scheduling – Partitioning

## ■ Static Partitioning

- $M = K$  tasks, task size  $\text{ceil}(N/K)$
- Low overhead, poor load balance

## ■ Fixed Partitioning

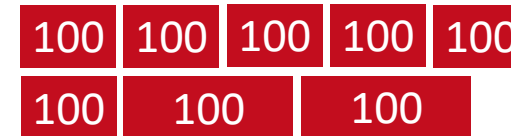
- $M = N/d$  tasks, task size  $d$
- E.g., # iterations, # tuples to process

## ■ Self-Scheduling

- Exponentially decreasing task sizes  $d$   
→  $M = \log N$  tasks (w/ min task size)
- Low overhead and good load balance at end
- Guided self scheduling
- Factoring: waves of task w/ equal size

## Example Hyper-param Tuning

```
parfor(i in 1:800)  
  R[i,] = lm(X,y,reg[i])
```



[Susan Flynn Hummel, Edith Schonberg, Lawrence E. Flynn: Factoring: a practical and robust method for scheduling parallel loops. **SC 1991**]



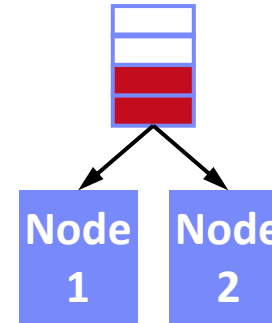


# Task Scheduling – Placement

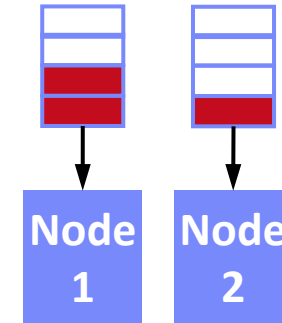


## ■ Task Queues

- Sequence of tasks in FIFO queue
- #1 **Single Task Queue**  
(self-balancing, but contention)
- #2 **Per-Worker Task Queue**  
(work separation, and preparation)



“Airport”



“Super Market”

## ■ Work Stealing

- On **empty worker queue**, probe other queues and “steal” tasks
- More common in multi-threading, difficult in distributed systems

## ■ Excursus: Power of 2 Choices

- Choose  $d$  bins at random, task in least full bin
- Reduce max load from  $\frac{\log M}{\log \log M}$  to  $\frac{\log \log M}{\log M}$

[Michael D. Mitzenmacher: The Power of Two Choices in Randomized Load Balancing, PhD Thesis UC Berkeley 1996]



# Spark Task Scheduling



## ■ Overview

- Schedule job DAGs in stages (shuffle barriers)
- Default task scheduler: **FIFO**; alternative: **FAIR**

## SystemDS Example (80GB):

```
X = rand(rows=1e7,cols=1e3)
parfor(i in 1:4)
  for(j in 1:10000)
    print(sum(X)) #spark job
```

### FIFO

Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Rea
37	<a href="#">fold at RDDAggregateUtils.java:150</a>	+details (kill)	2019/12/12 23:48:07	Unknown	0/596			
36	<a href="#">fold at RDDAggregateUtils.java:150</a>	+details (kill)	2019/12/12 23:48:06	0.7 s	391/596 (23 running)	48.9 GB		
35	<a href="#">fold at RDDAggregateUtils.java:150</a>	+details (kill)	2019/12/12 23:48:05	1 s	424/596 (20 running)	53.0 GB		
34	<a href="#">fold at RDDAggregateUtils.java:150</a>	+details (kill)	2019/12/12 23:48:05	2 s	504/596 (20 running)	63.0 GB		

### FAIR

#### Fair Scheduler Pools (5)

Pool Name	Minimum Share	Pool Weight	Active Stages	Running Tasks	SchedulingMode
default	0	1	0	0	FIFO
<a href="#">parforPool2</a>	0	1	1	38	FIFO
<a href="#">parforPool1</a>	0	1	1	16	FIFO
<a href="#">parforPool3</a>	0	1	1	3	FIFO
<a href="#">parforPool0</a>	0	1	1	43	FIFO

#### Active Stages (4)

Stage Id ▾	Pool Name	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Rea
206	<a href="#">parforPool0</a>	<a href="#">fold at RDDAggregateUtils.java:150</a>	+details (kill)	2019/12/12 23:14:20	1.0 s	368/596 (67 running)	46.0 GB		
205	<a href="#">parforPool2</a>	<a href="#">fold at RDDAggregateUtils.java:150</a>	+details (kill)	2019/12/12 23:14:20	1 s	432/596 (43 running)	54.0 GB		
204	<a href="#">parforPool1</a>	<a href="#">fold at RDDAggregateUtils.java:150</a>	+details (kill)	2019/12/12 23:14:19	2 s	561/596 (11 running)	70.1 GB		
203	<a href="#">parforPool3</a>	<a href="#">fold at RDDAggregateUtils.java:150</a>	+details (kill)	2019/12/12 23:14:19	2 s	590/596 (6 running)	73.7 GB		

# Spark Task Scheduling, cont.

- FAIR scheduling  
w/ k=32  
concurrent jobs  
and 200GB

**FAIR:**  
Share 320 cores  
among 32  
concurrent jobs  
→ ~10 tasks/job

Elapsed:  
~40min

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(11)	1490	200 GB / 595.3 GB	0.0 B	320	329	0	8714054	8714383	218.4 h (57 min)	1.2 PB	0.0 B	0.0 B	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(11)	1490	200 GB / 595.3 GB	0.0 B	320	329	0	8714054	8714383	218.4 h (57 min)	1.2 PB	0.0 B	0.0 B	0

## Active Stages (32)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
663	parforPool7	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:58	0.3 s	48/1490 (25 running)	6.0 GB			
662	parforPool9	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:57	0.7 s	186/1490 (25 running)	23.3 GB			
661	parforPool10	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:57	0.7 s	221/1490 (24 running)	27.6 GB			
660	parforPool11	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:57	0.8 s	327/1490 (25 running)	40.9 GB			
659	parforPool21	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:57	2 s	506/1490 (9 running)	63.3 GB			
658	parforPool6	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:56	2 s	518/1490 (9 running)	64.8 GB			
657	parforPool1	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:56	2 s	572/1490 (10 running)	71.5 GB			
656	parforPool24	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:56	3 s	603/1490 (9 running)	75.4 GB			
655	parforPool13	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:55	3 s	684/1490 (10 running)	85.5 GB			
654	parforPool20	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:54	4 s	736/1490 (10 running)	92.0 GB			
653	parforPool4	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:54	4 s	750/1490 (9 running)	93.8 GB			
652	parforPool23	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:54	5 s	797/1490 (7 running)	99.6 GB			
651	parforPool15	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:53	5 s	847/1490 (9 running)	105.9 GB			
650	parforPool29	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:53	5 s	808/1490 (9 running)	101.0 GB			
649	parforPool2	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:52	6 s	926/1490 (9 running)	115.8 GB			
648	parforPool26	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:52	6 s	917/1490 (9 running)	114.6 GB			
647	parforPool31	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:52	6 s	913/1490 (9 running)	114.1 GB			
646	parforPool19	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:51	7 s	1023/1490 (9 running)	127.9 GB			
645	parforPool5	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:51	7 s	1011/1490 (7 running)	126.4 GB			
644	parforPool30	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:50	8 s	1036/1490 (9 running)	129.5 GB			
643	parforPool3	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:49	9 s	1056/1490 (8 running)	132.0 GB			
642	parforPool17	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:49	9 s	1125/1490 (9 running)	140.6 GB			
641	parforPool16	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:49	9 s	1158/1490 (9 running)	144.7 GB			
640	parforPool18	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:49	9 s	1124/1490 (9 running)	140.5 GB			
639	parforPool0	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:48	10 s	1287/1490 (9 running)	160.9 GB			
638	parforPool28	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:48	10 s	1251/1490 (9 running)	156.4 GB			
637	parforPool12	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:48	11 s	1341/1490 (9 running)	167.6 GB			
636	parforPool27	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:47	12 s	1309/1490 (9 running)	163.6 GB			
635	parforPool8	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:47	12 s	1299/1490 (8 running)	162.4 GB			
634	parforPool14	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:46	12 s	1413/1490 (9 running)	176.6 GB			
633	parforPool25	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:46	12 s	1343/1490 (9 running)	167.9 GB			
632	parforPool22	fold at RDDAggregateUtils.java:148	2021/11/27 15:51:46	12 s	1415/1490 (7 running)	176.9 GB			



# Spark Task Scheduling, cont.



## ■ Fair Scheduler Configuration

- Pools with shares of cluster
- Scheduling modes: FAIR, FIFO
- **weight**: relative to equal share
- **minShare**: min numCores

```
<allocations>
  <pool name="data_science">
    <schedulingMode>FAIR</schedulingMode>
    <weight>1</weight> <minShare>6</minShare>
  </pool>
  <pool name="indexing">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight> <minShare>8</minShare>
  </pool>
</allocations>
```

## ■ Spark on Kubernetes

- Run Spark in shared cluster with Docker container apps, Distributed TensorFlow, etc
- Custom controller, and shuffle service (dynAlloc)

```
$SPARK_HOME/bin/spark-submit \
--master k8s://https://<k8s-api>:<k8s-api-port> \
--deploy-mode cluster
--driver-java-options "-server -Xms40g -Xmn4g" \
--driver-memory 40g \
--num-executors 10 \
--executor-memory 100g \
--executor-cores 32 \
--conf spark.kubernetes.container.image=<sparkimg> \
SystemDS.jar -f test.dml -stats -explain -args ...
```

## ■ Configuration for YARN/Mesos

- Set `spark.dynamicAllocation.enabled = true`
- Set `spark.shuffle.service.enabled = true` (robustness w/ stragglers)

## ■ Executor Addition/Removal

- **Approach:** look at task pressure (pending tasks / idle executors)
- Increase exponentially (add **1, 2, 4, 8**) if  
pending tasks for `spark.dynamicAllocation.schedulerBacklogTimeout`
- Decrease executors they are idle for `spark.dynamicAllocation.executorIdleTimeout`

```
spark-submit \  
  --conf spark.shuffle.service.enabled=true \  
  --conf spark.dynamicAllocation.enabled=true \  
  --conf spark.dynamicAllocation.minExecutors=0 \  
  --conf spark.dynamicAllocation.initialExecutors=1 \  
  --conf spark.dynamicAllocation.maxExecutors=20
```

# Sparrow Task Scheduling

[Kay Ousterhout, Patrick Wendell, Matei Zaharia, Ion Stoica: **Sparrow: distributed, low latency scheduling**. **SOSP 2013**]

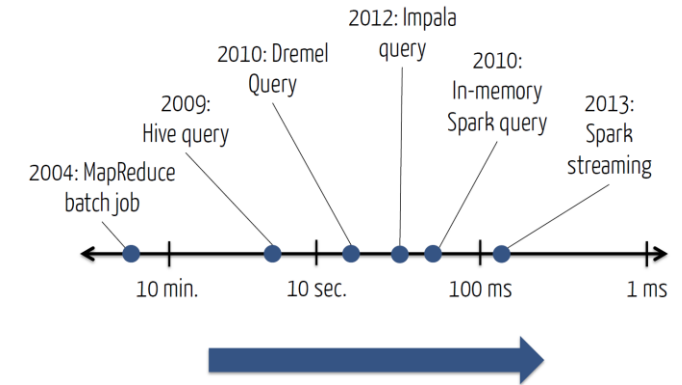


## ■ Sparrow Overview

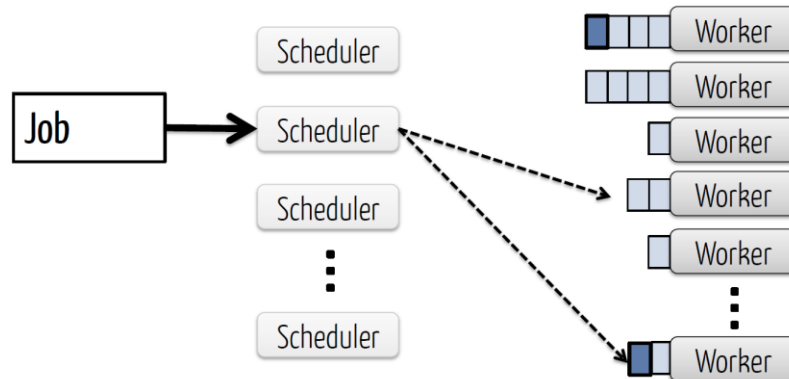
- Decentralized, randomized task scheduling with constraints, fair sharing
- Problems:** Low latency, quality placement, fault tolerance, high throughput

## ■ Approach

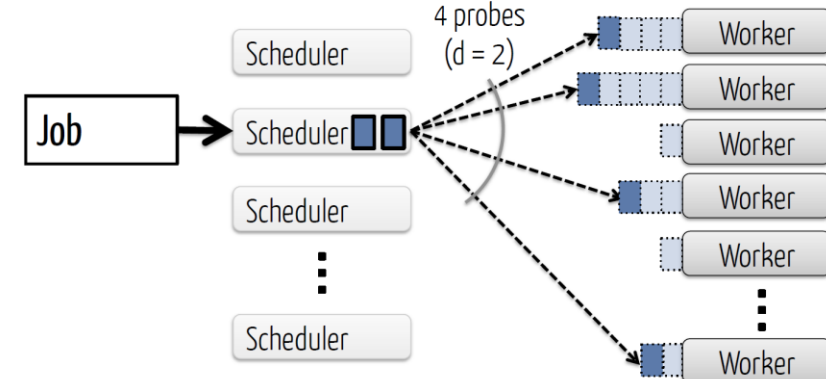
- Baselines:** Random, Per-task (power of two choices)
- New Techniques: Batch Scheduling, Late Binding



### Baseline: Per-task sampling



### Batch sampling w/ late binding



# Resource Elasticity in SystemML

[Botong Huang et al.: Resource Elasticity for Large-Scale Machine Learning. **SIGMOD 2015**]

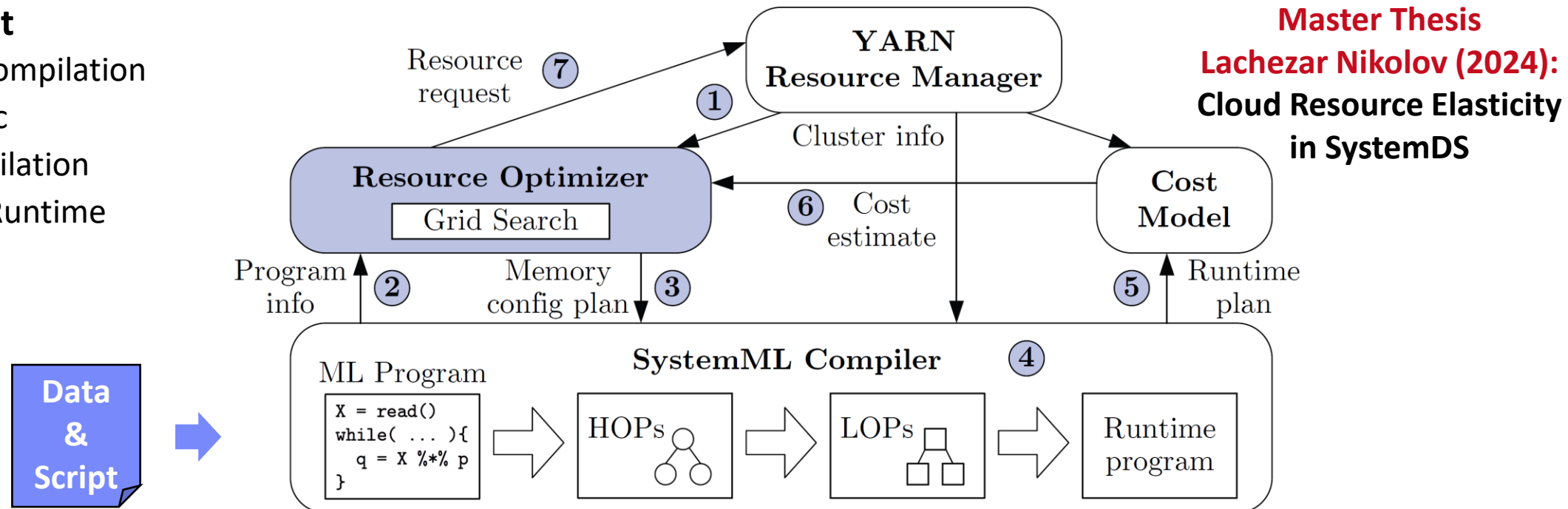


## ■ Resource Optimizer for ML Workloads

- Optimize ML program resource configurations via online **what-if analysis and plan generation**
- **Minimize cost w/o unnecessary overprovisioning**, program-aware enumeration (e.g., mem estimates)

## ■ Deployment

- Initial Compilation
- Dynamic Recompilation during Runtime



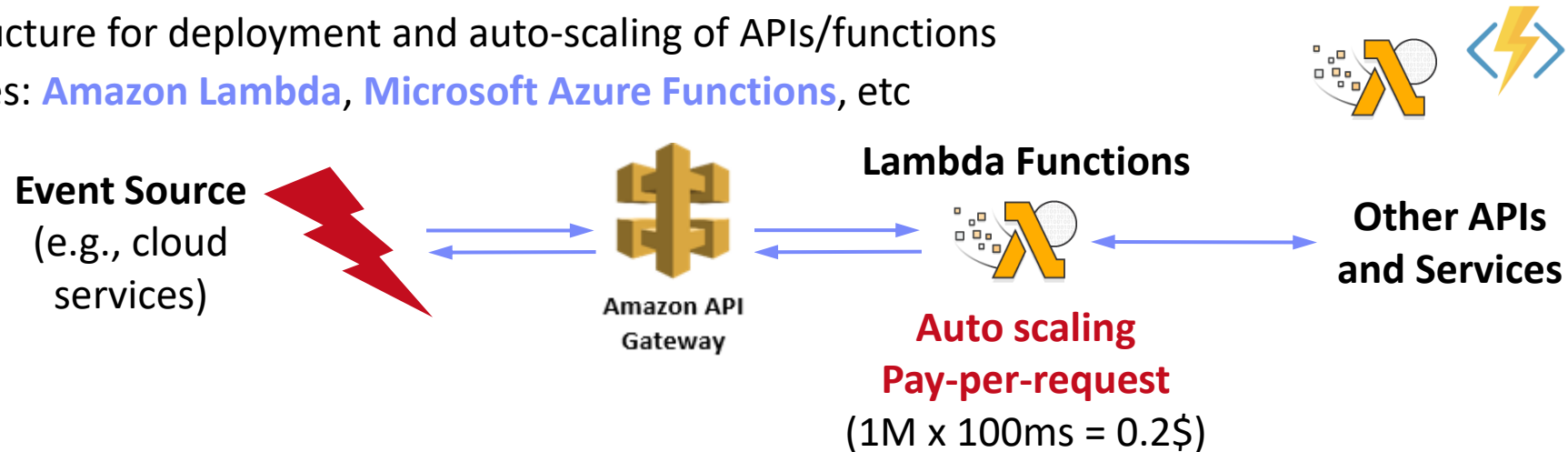
# Serverless Computing (FaaS)

[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



## ■ 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 expensiveModelScoring(input); // with read-only model
    }
}
```



# Summary and Q&A



- Motivation, Terminology, and Fundamentals
- Resource Allocation, Isolation, and Monitoring
- Task Scheduling and Elasticity
- Next Lectures (**Large-scale Data Management and Analysis**)
  - 10 **Distributed Data Storage** [Dec 18]
  - **Holidays**
  - 11 **Distributed, Data-Parallel Computation** [Jan 15]
  - 12 **Distributed Stream Processing** [Jan 22]
  - 13 **Distributed Machine Learning Systems** [Jan 29]