Data Integration and Large Scale Analysis
08 Cloud Resource Management

Shafaq Siddiqi

Graz University of Technology, Austria

Last update: Nov 25, 2022

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

12 Distributed Stream Processing

13 Distributed Machine Learning Systems

Compute/ Storage

11 Distributed Data-Parallel Computation

10 Distributed Data Storage

Infra

09 Cloud Resource Management and Scheduling

08 Cloud Computing Fundamentals

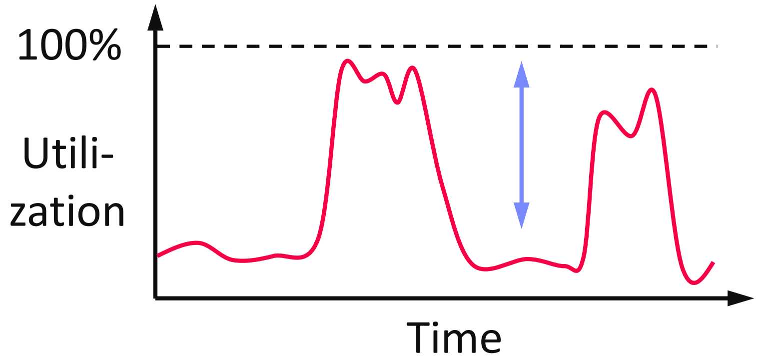
Agenda

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

Motivation, Terminology, and Fundamentals

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**

- **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

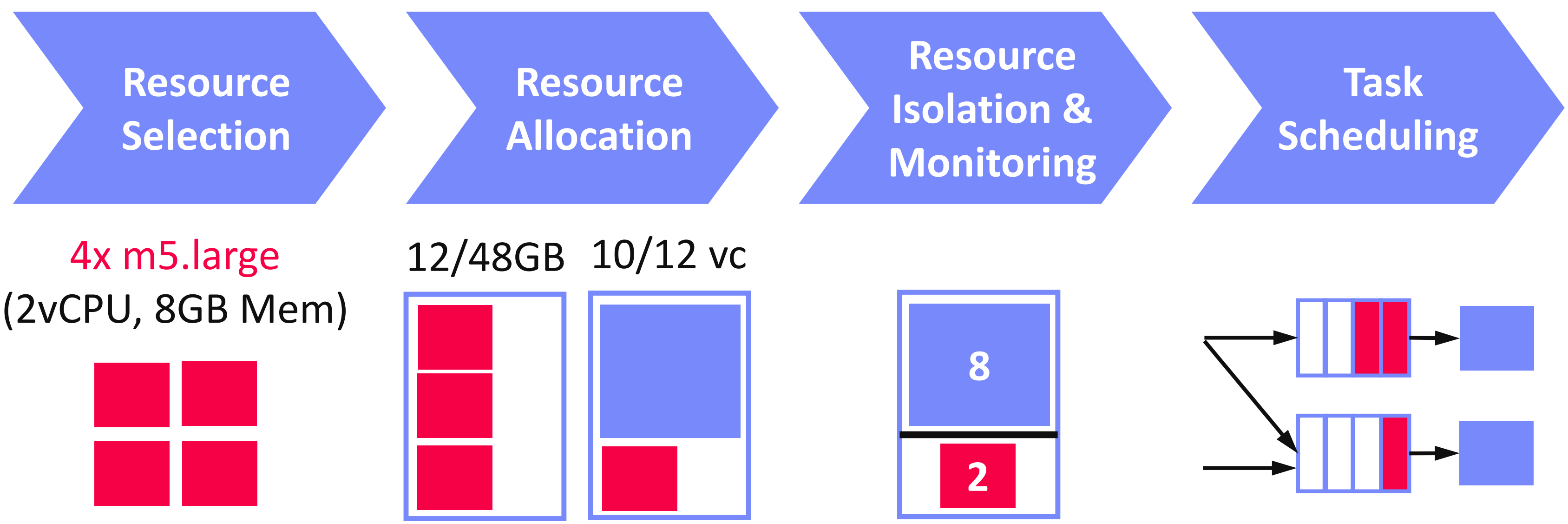
(but beware Amdahl's law: max speedup **sp = 1/s**)

Overview Resource Management & Scheduling

- **Resource Bundles**
  - Logical containers (aka nodes/instances) of different resources (**vcores**, **mem**)
  - Disk capacity, **disk** and **network** bandwidth
  - Accelerator devices (**GPUs**, FPGAs), etc

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

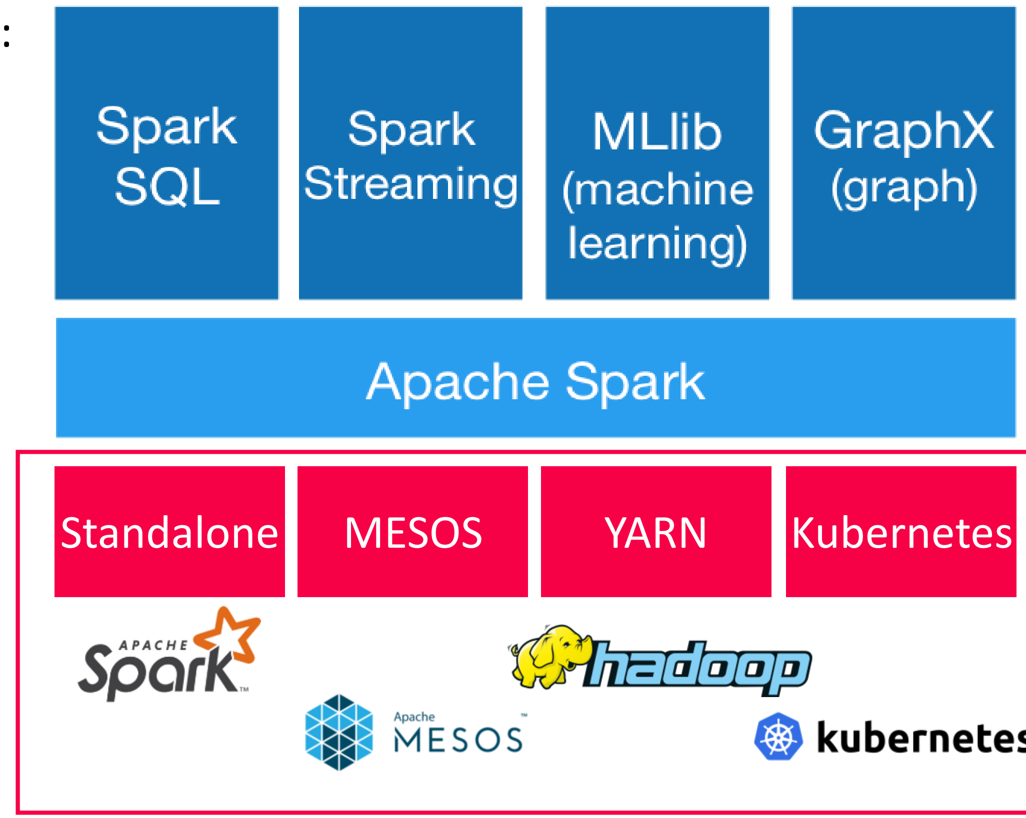
- **Resource Management**



Recap: Apache Spark History and Architecture

- **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**

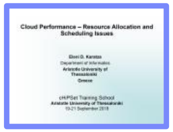
[https://spark.apache.org/]



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

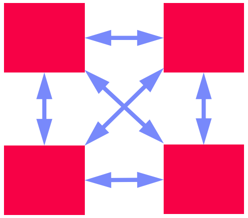
Scheduling Problems

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



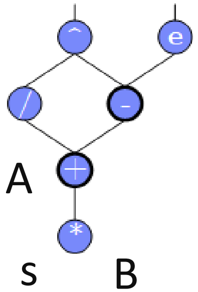
- **Bag-of-Tasks Scheduling**
  - Job of **independent** (embarrassingly parallel) tasks
  - **Examples:** EC2 instances, map tasks

- **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

$$\mathbf{C} = \mathbf{A} + s * \mathbf{B}$$
$$\mathbf{D} = (\mathbf{C}/2)\hat{}(\mathbf{C}\text{-}1)$$
$$\mathbf{E} = \exp(\mathbf{C}\text{-}1)$$



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

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)
- **#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

Resource Allocation, Isolation, and Monitoring

Resource Selection

- **#1 Manual Selection**
  - Rule of thumb (I/O, mem, CPU characteristics of app)
  - Data characteristics, and framework configurations, experience

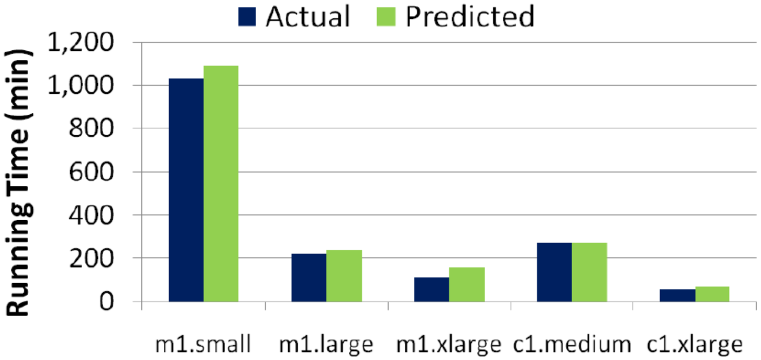
- **Example Spark Submit**

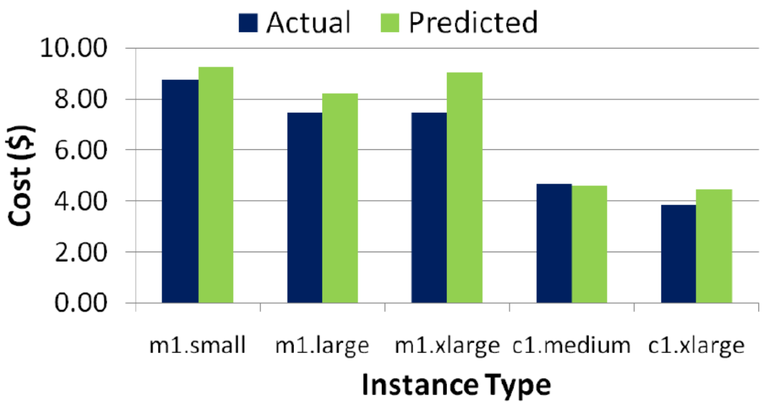
```
export HADOOP_CONF_DIR=/etc/hadoop/conf
SPARK_HOME=../spark-2.4.0-bin-hadoop2.7

$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)
  - Min $costs under time constraint
  - Min runtime under $cost constraint





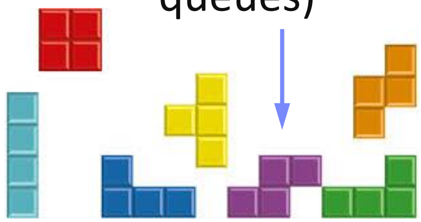
[Herodotos Herodotou, Fei Dong, Shivnath Babu: No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. **SoCC 2011**]

(fixed MR job w/ 6 nodes)

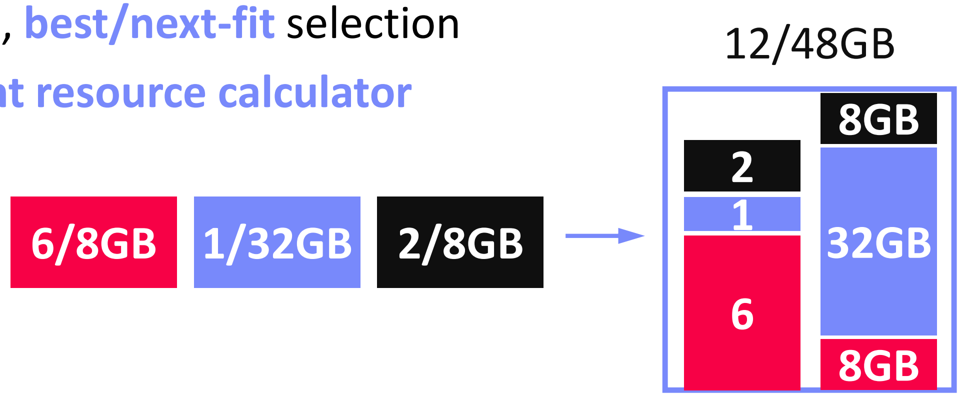
Resource Negotiation and Allocation

- **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**)

"Tetris Analogy" (w/ expiration and queues)



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



[https://blog.cloudera.com/managing-cpu-resources-in-your-hadoop-yarn-clusters/]

Slurm Workload Manager

- **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

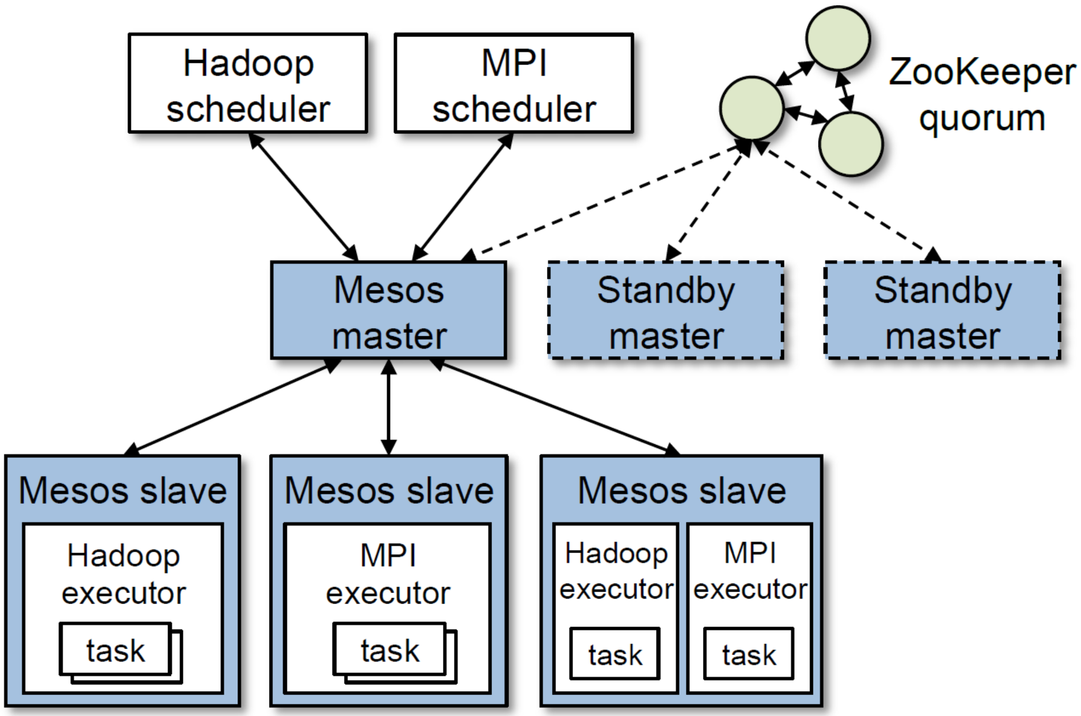
[Matei Zaharia et al: Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. **EuroSys 2010**]

- **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**]

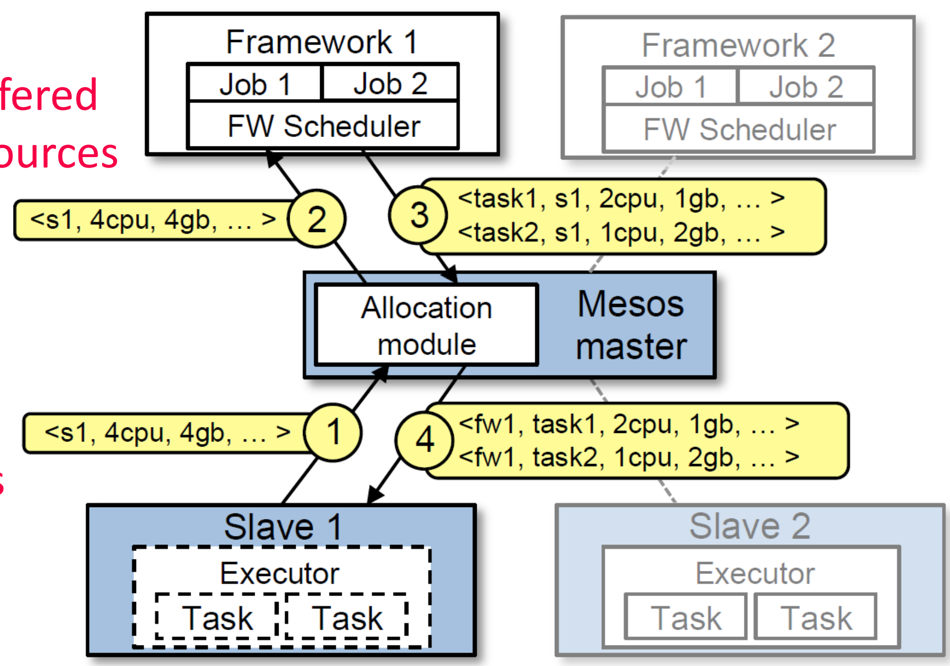
- **Overview Mesos**
  - Fine-grained, **multi-framework cluster sharing**
  - Scalable and efficient scheduling → **delegated to frameworks**
  - **Resource offers**



Mesos Resource Management, cont.

- **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**

Offered resources



Reported available resources

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

YARN Resource Management

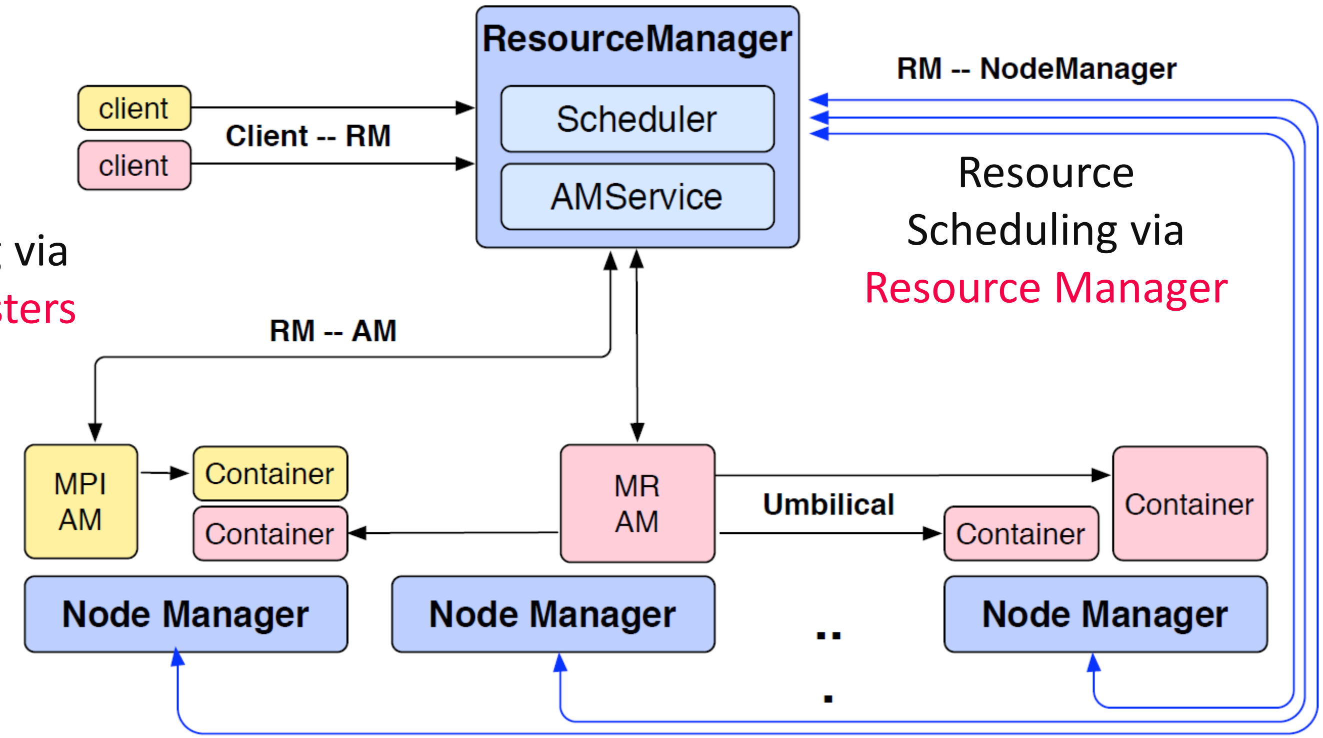
[Vinod Kumar Vavilapalli et al: Apache Hadoop YARN: yet another resource negotiator. **SoCC 2013**]

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

Task Scheduling via Application Masters (AMs)

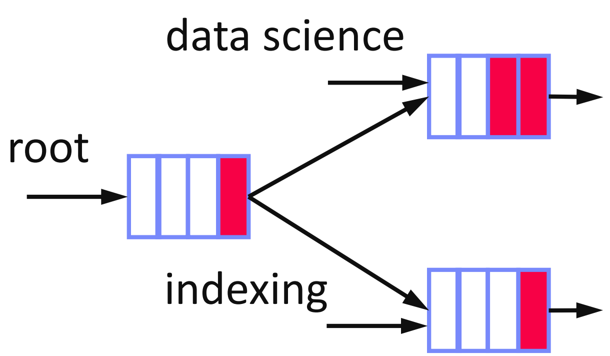
Resource Scheduling via Resource Manager

Resource Isolation via Node Managers



YARN Resource Management, cont.

- **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)



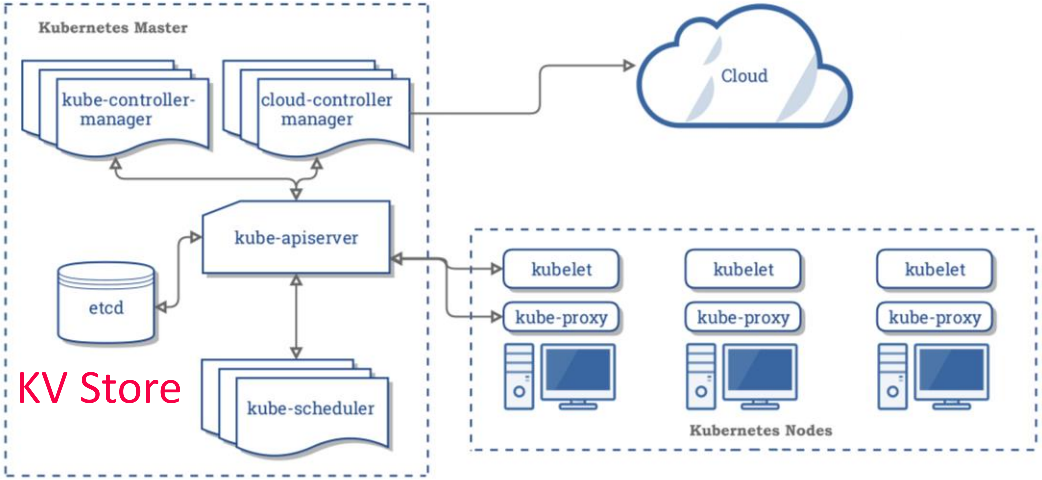
- **Fair Scheduler**
  - All applications get same resources over time
  - Fairness decisions on memory requirements, but dominant resource fairness possible too

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/]

Kubernetes Container Orchestration, cont.

- **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, percentageOfNodesToScore [1,100])`
    (sample taken round robin across zones)

  ➔ **Binding:** scheduler notifies API server

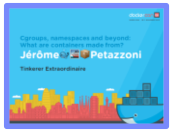
Resource Isolation

- **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, name-spaces and beyond: What are containers made from? DockerConEU 2015.]

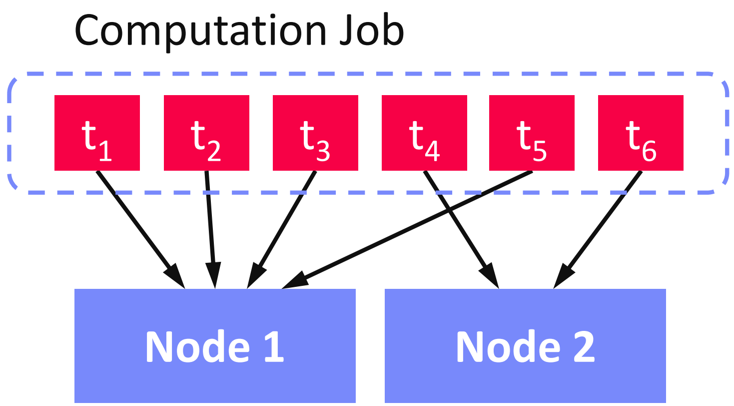


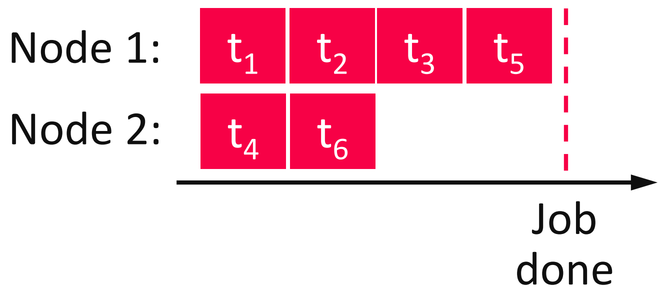
[https://www.youtube.com/watch?v=sK5i-N34im8&feature=youtu.be]

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



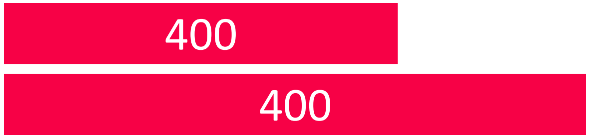


Task Scheduling – Partitioning

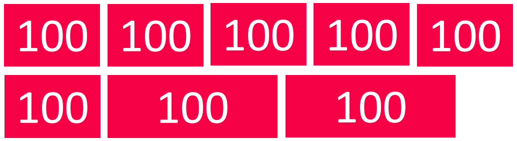
Example Hyper-param Tuning

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

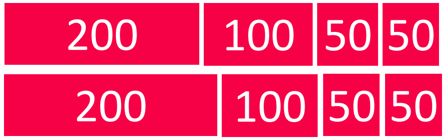
- **Static Partitioning**
  - M = K tasks, task size ceil(N/K)
  - Low overhead, poor load balance



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



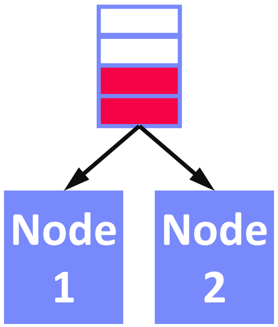
- **Self-Scheduling**
  - Exponentially decreasing task sizes d
    → M = log N tasks (w/ min task size)
  - Low overhead and good load balance at end
  - **Guided self scheduling**
  - **Factoring:** waves of task w/ equal size



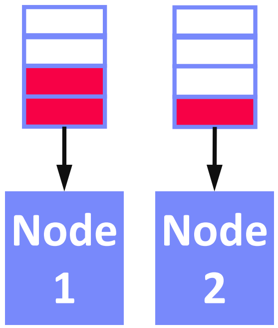
[Susan Flynn Hummel, Edith Schonberg, Lawrence E. Flynn: Factoring: a practical and robust method for scheduling parallel loops. **SC 1991**]

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	fold at RDDAggregateUtils.java:150	+details	(kill)	2019/12/12 23:48:07	Unknown	0/596			
36	fold at RDDAggregateUtils.java:150	+details	(kill)	2019/12/12 23:48:06	0.7 s	391/596 (23 running)	48.9 GB		
35	fold at RDDAggregateUtils.java:150	+details	(kill)	2019/12/12 23:48:05	1 s	424/596 (20 running)	53.0 GB		
34	fold at RDDAggregateUtils.java:150	+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
parforPool2	0	1	1	38	FIFO
parforPool1	0	1	1	16	FIFO
parforPool3	0	1	1	3	FIFO
parforPool0	0	1	1	43	FIFO

Active Stages (4)

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

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

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

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 …
```

Spark Dynamic Allocation

[https://spark.apache.org/docs/latest/job-scheduling.html]

- **Configuration for YARN/Mesos**
  - Set `spark.dynamicAllocation.enabled = true`
  - Set `spark.shuffle.service.enabled = true` (robustness w/ stragglers)

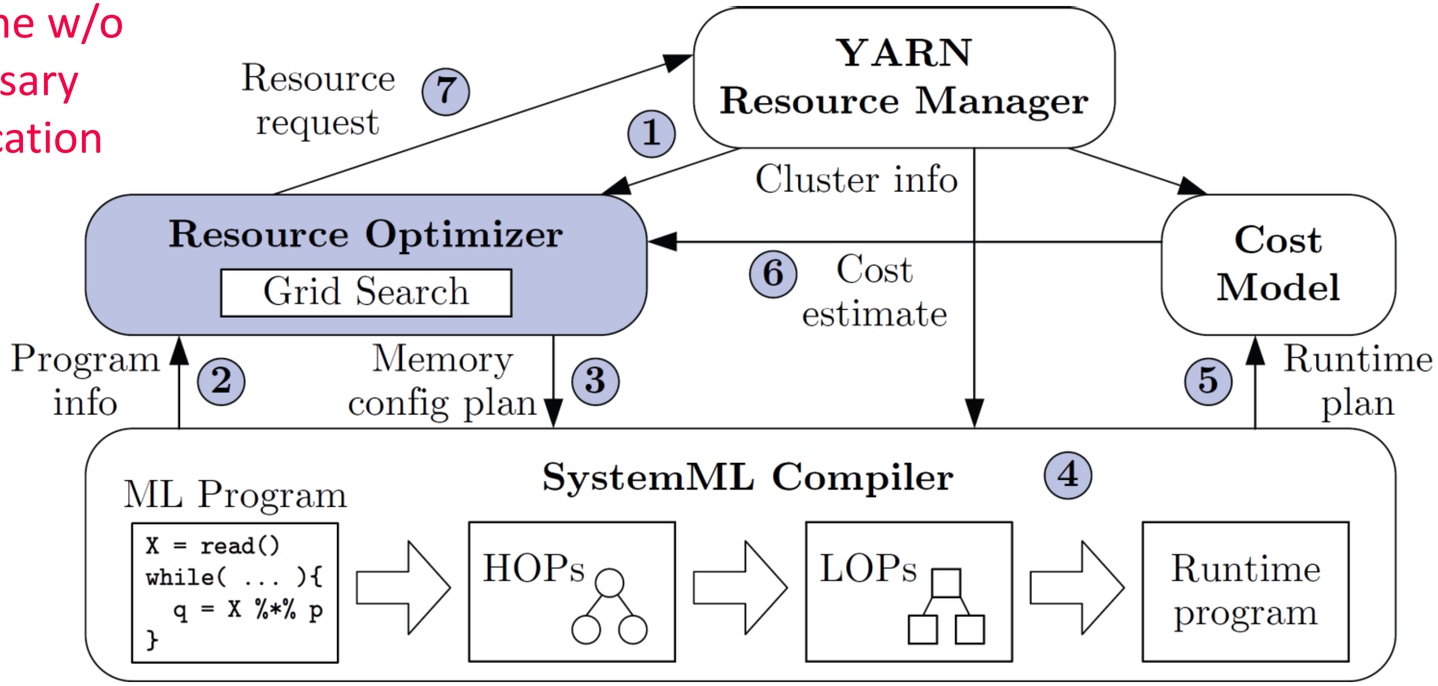
- **Executor Addition/Removal**
  - **Approach:** look at task pressure (pending tasks / idle executors)
  - Increase exponentially (add **1**, **2**, **4**, **8**) if pending tasks for `spark.dynamicAllocation.schedulerBacklogTimeout`
  - Decrease executors they are idle for `spark.dynamicAllocation.executorIdleTimeout`

Resource Elasticity in SystemML

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

- **Basic Ideas**
  - Optimize ML program resource configurations via **online what-if analysis**
  - Generating and **costing runtime plans** for local/MR
  - **Program-aware** grid enumeration, pruning, and re-optimization techniques

Min runtime w/o unnecessary over-allocation



Summary and Q&A

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

- **Next Lectures**
  - **10 Distributed Data Storage** [Dec 02]