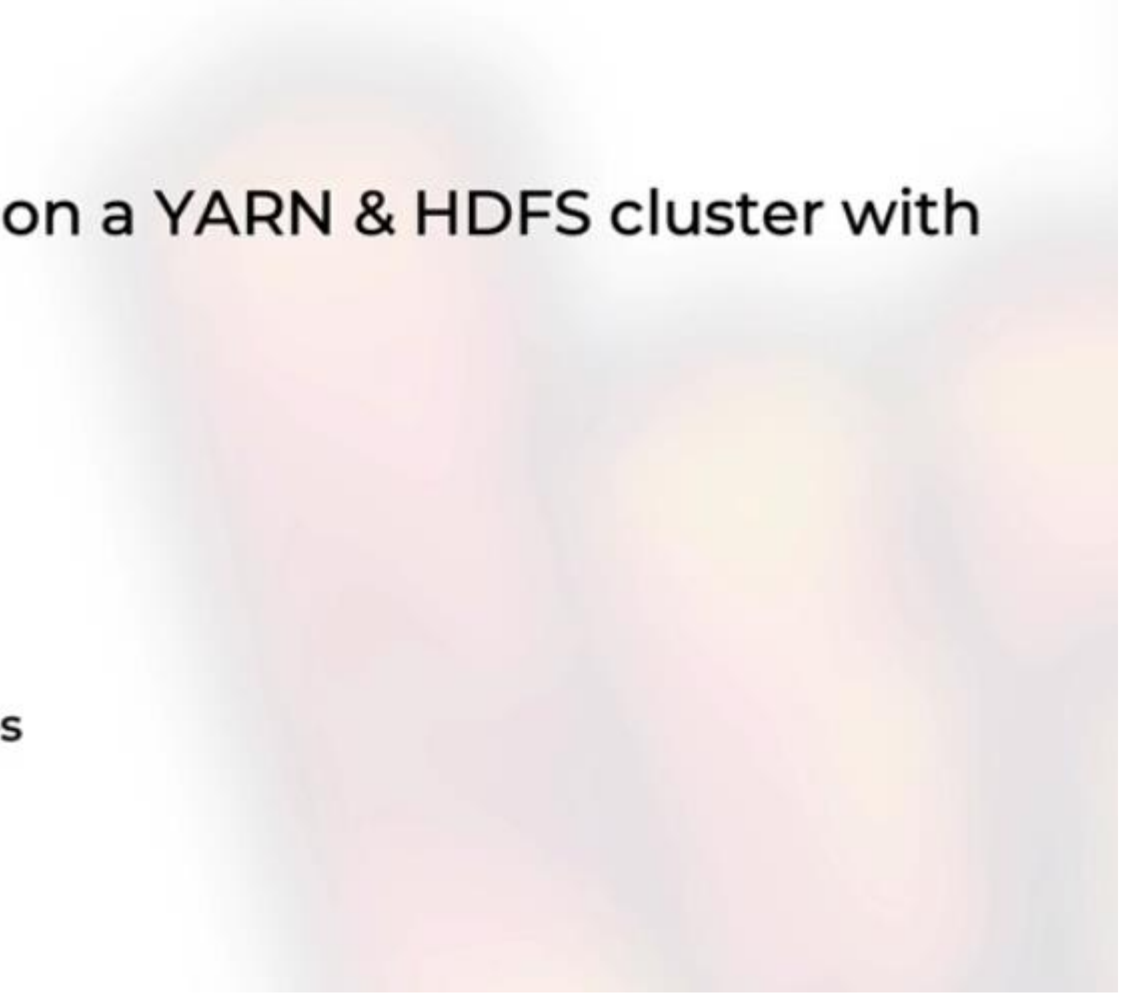
Resource Planning

Example: let's tune a Spark job running on a YARN & HDFS cluster with

- 4 nodes
- 64GB each
- 16 cores each



Fat & Skinny executors problem:

- Skinny executors = lots of executors with few resources
- Fat executors = few executors with lots of resources
- Fit executors = just right

Resource Planning

Scenario 1: skinny executors

- 64 executors
- 1 core each
- 4GB each

```
spark.executor.memory = 4g
spark.executor.cores = 1
spark.executor.instances = 64
```

```
spark-submit --num-executors 64 --executor-memory 4g --executor-cores 1
```

Pros

- good I/O throughput
- (maybe) good for lots of small tasks

Cons

- each executor is single-threaded
- bigger tasks will OOM the executors
- managing so many incurs a large overhead

Resource Planning

Scenario 2: fat executors

- 4 executors, one per machine
- 16 cores each
- 64GB each

```
spark.executor.memory = 64g
spark.executor.cores = 16
spark.executor.instances = 4
```

```
spark-submit --num-executors 4 --executor-memory 64g --executor-cores 16
```

Pros

- can accommodate enormous tasks
- can leverage executor parallelism

Cons

- bad for HDFS and concurrent I/O
- 64GB won't fit in the container mem
- one executor down = 25% of cluster down

Resource Planning

Scenario 3: fit executors

- allocate CPU & mem for YARN, OS and HDFS, e.g. 4GB and 4 cores per machine
- left: 60GB and 12 cores, x4 machines
- allocate ~1GB for ApplicationMaster (not significant for large clusters)
- keep <= 5 cores/executor for good HDFS throughput
- 48/5 = 9 executors
- memory/executor = (4 x 60 – 1) / 9 = 26GB
- keep 7-8% for executor overhead
- => net executor memory 24GB

```
spark-submit --num-executors 9 --executor-memory 24g --executor-cores 5
```

Pros

- good I/O throughput
- can handle large tasks
- can handle lots of tasks
- leverage executor parallelism

Cons

- 9 executors on 4 machines is not 100% even

Resource Planning

Exercise: an HDFS & YARN cluster with

- 10 machines
- 64GB RAM and 16CPUs per machine

Solution:

- keep 4 cores and 4GB RAM per machine => 40 CPUs and 40 GB RAM for YARN, OS, daemons
- remaining: 120 CPUs, 600 GB RAM
- AM is negligible, will take down from executor mem
- 5 cores / executor => 24 executors
- memory/executor = 600/24 = 25 GB
- minus ~8% overhead => net executor memory 22-23 GB

```
spark-submit --num-executors 24 --executor-memory 22g --executor-cores 5
```

Resource Planning

Exercise: a large HDFS & YARN cluster on AWS with

- 1 master node r5.12xlarge
- 19 r5.12xlarge worker nodes
- 8 TB total RAM
- 960 total virtual CPUs

Solution:

- leave the master node alone – will keep AM
- keep 4 cores and 4GB RAM per machine => 76 cores & 76GB
- remaining: ~7920 GB RAM, 884 cores
- 5 cores / executor => 176 executors
- memory/executor = 7920/176 = 45 GB
- minus 8% overhead => net executor memory ~41 GB

```
spark-submit --num-executors 176 --executor-memory 41g --executor-cores 5
```

Dynamic Resource Allocation

Allows Spark to request/terminate executors as the job is running

- useful for multi-tenancy/cluster sharing/when cluster is idle
- maximize cluster utilization
- good for long-running jobs
- can impact performance for low-latency jobs

Configs:

```
spark.dynamicAllocation.enabled
spark.dynamicAllocation.initialExecutors
spark.dynamicAllocation.minExecutors
spark.dynamicAllocation.maxExecutors
spark.dynamicAllocation.schedulerBacklogTimeout
spark.dynamicAllocation.sustainedSchedulerBacklogTimeout
spark.dynamicAllocation.executorIdleTimeout
```

requests new executor if task idle for longer than this time

Spark allocates executors in rounds

- requests new executors if task queue still not empty after current round
- exponential increases