SAPIENZA
UNIVERSITÀ DI ROMA

Hadoop Internals

Emilio Coppa

April 29, 2014

Big Data Computing
Master of Science in Computer Science

Hadoop Facts

- Open-source software framework for storage and large-scale processing of data-sets on clusters of commodity hardware.
- MapReduce paradigm: "Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages" (Dean && Ghemawat – Google – 2004)
- First released in 2005 by D. Cutting (Yahoo) and Mike Cafarella (U. Michigan)

Hadoop Facts (2)

- 2,5 millions of LOC – Java (47%), XML (36%)
- 681 years of effort (COCOMO)
- Organized in 4 projects: Common, HDFS, YARN, MapReduce
- 81 contributors

Hadoop Facts (3) – Top Contributors

Analyzing the top 10 of contributors...

Hadoop Facts (3) – Top Contributors

Analyzing the top 10 of contributors...

1. 6 HortonWorks ("We Do Hadoop")

Hadoop Facts (3) – Top Contributors

Analyzing the top 10 of contributors...

1. 6 HortonWorks ("We Do Hadoop")
2. 3 Cloudera ("Ask Big Questions")

Hadoop Facts (3) – Top Contributors

Analyzing the top 10 of contributors...

1. 6 HortonWorks ("We Do Hadoop")
2. 3 Cloudera ("Ask Big Questions")
3. 1 Yahoo

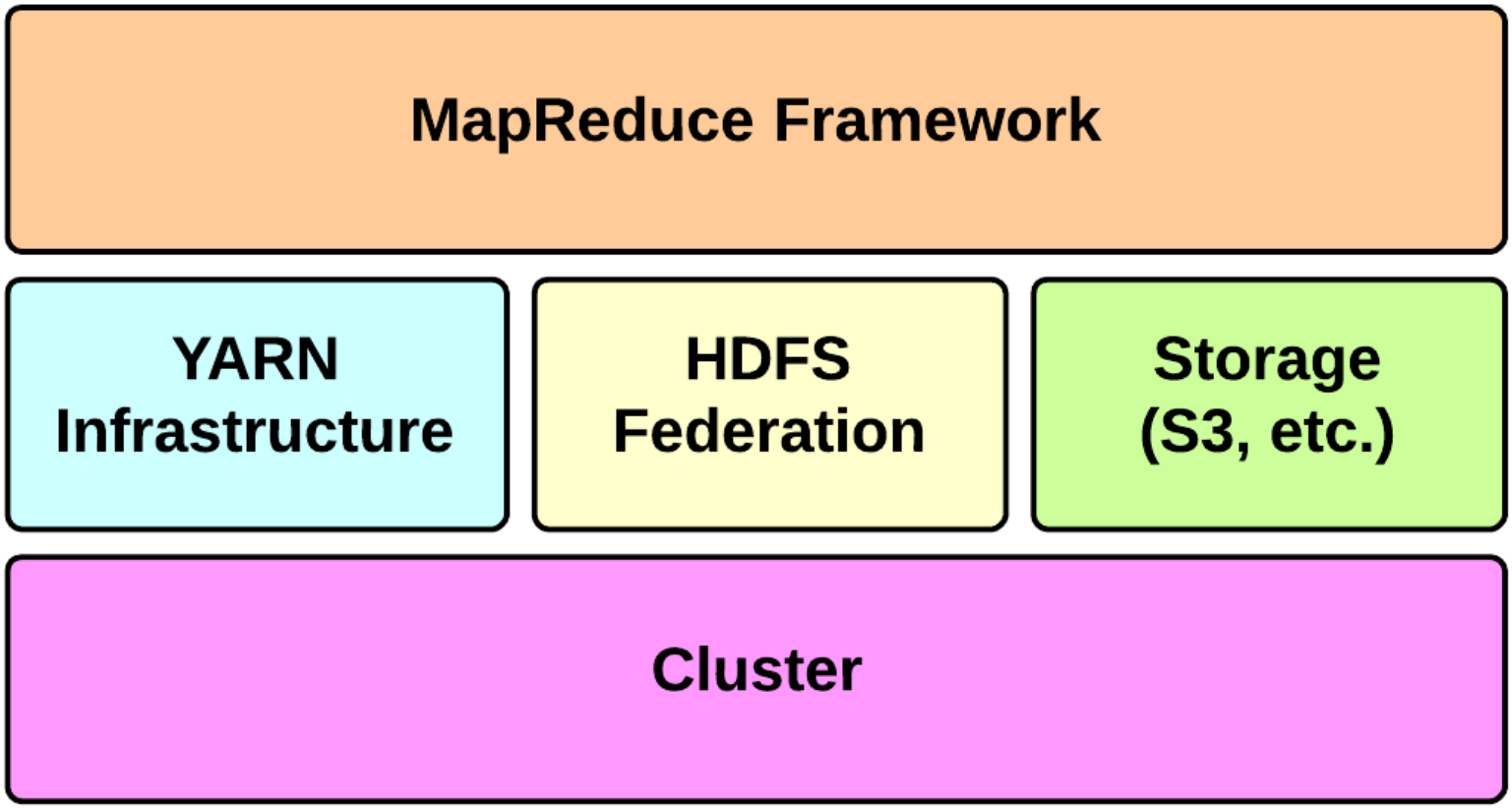
Hadoop Facts (3) – Top Contributors

Analyzing the top 10 of contributors...

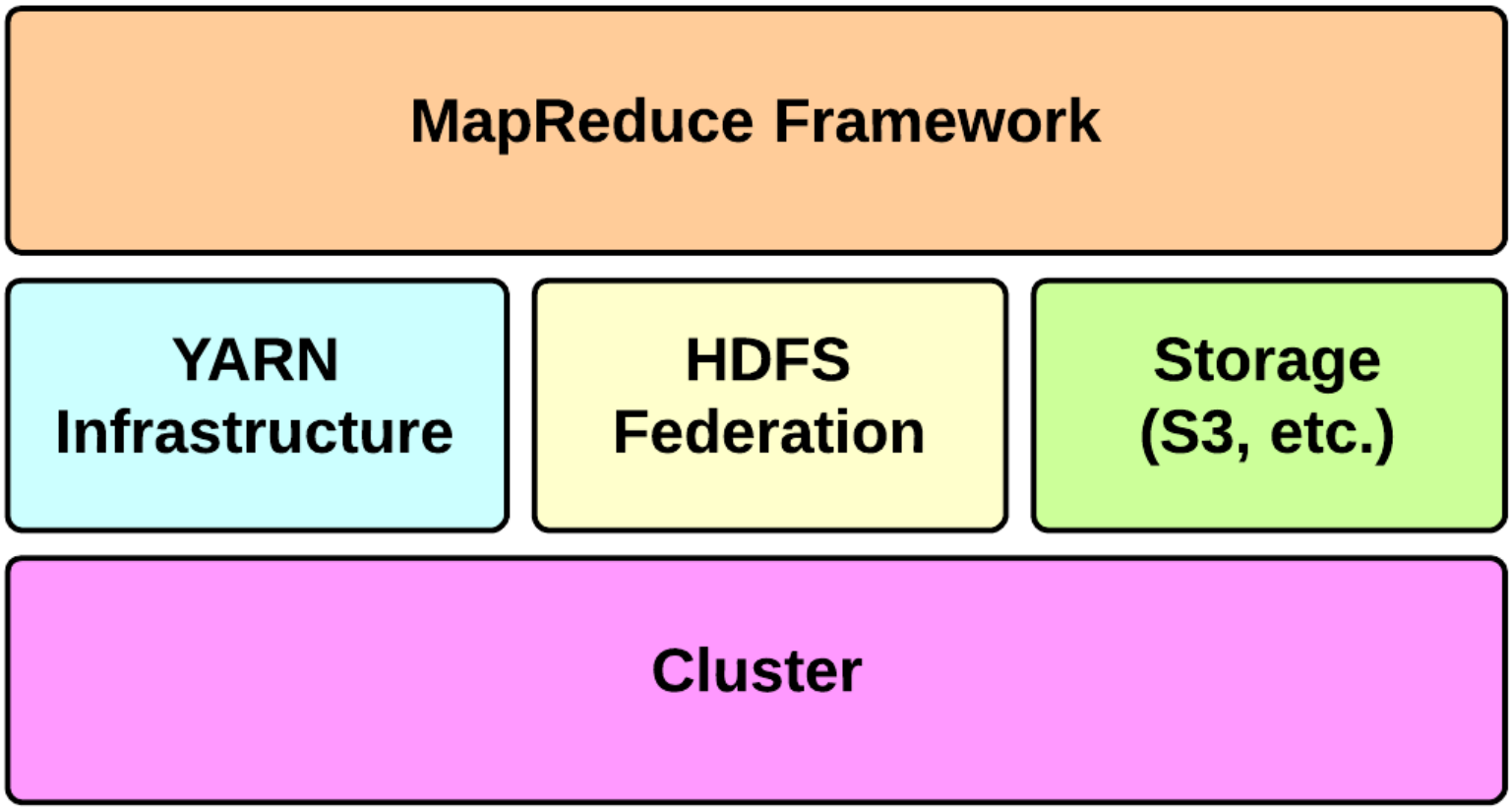
1. 6 HortonWorks ("We Do Hadoop")
2. 3 Cloudera ("Ask Big Questions")
3. 1 Yahoo

Doug Cutting currently works at Cloudera.

Apache Hadoop Architecture

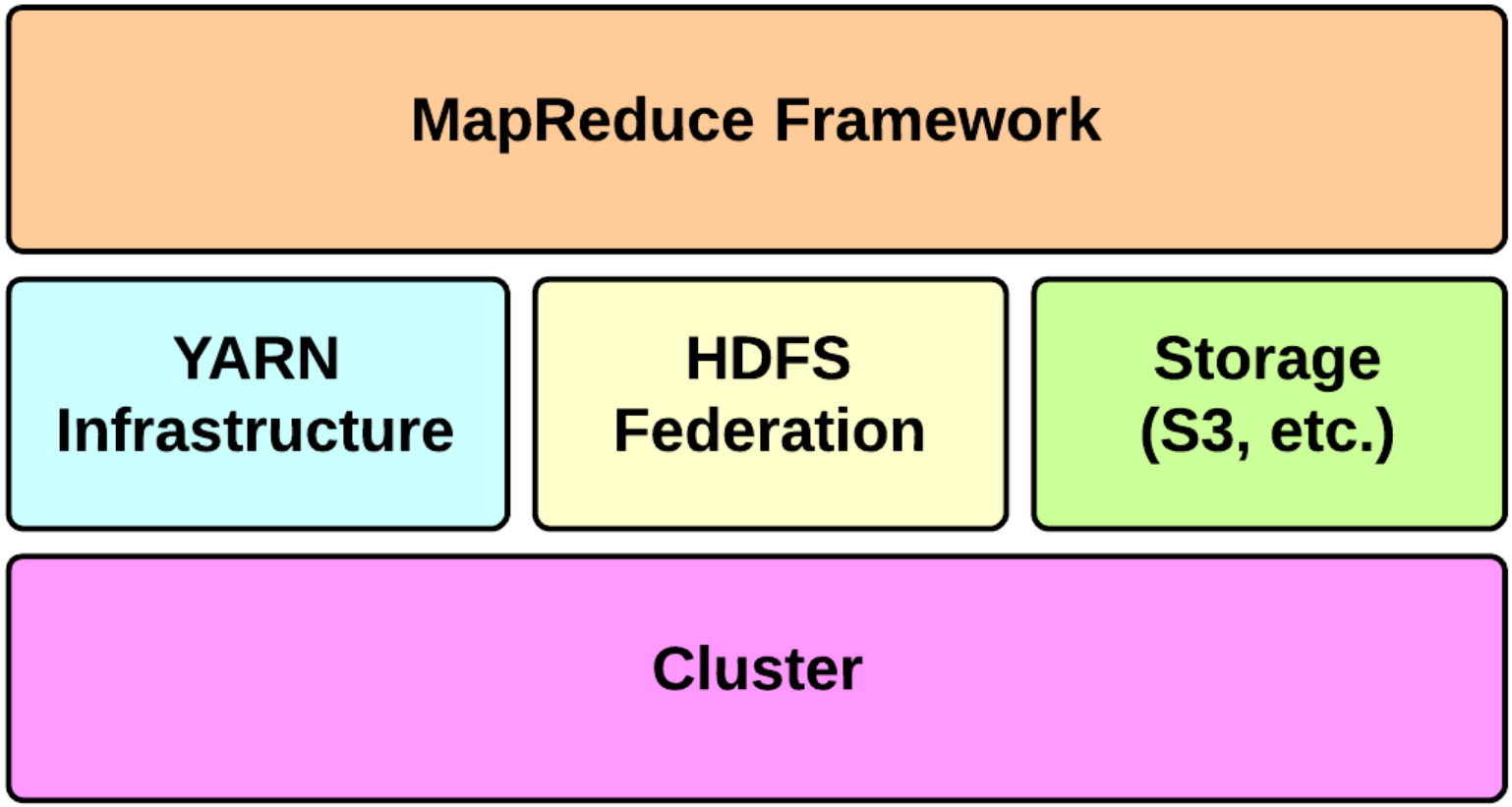


Apache Hadoop Architecture



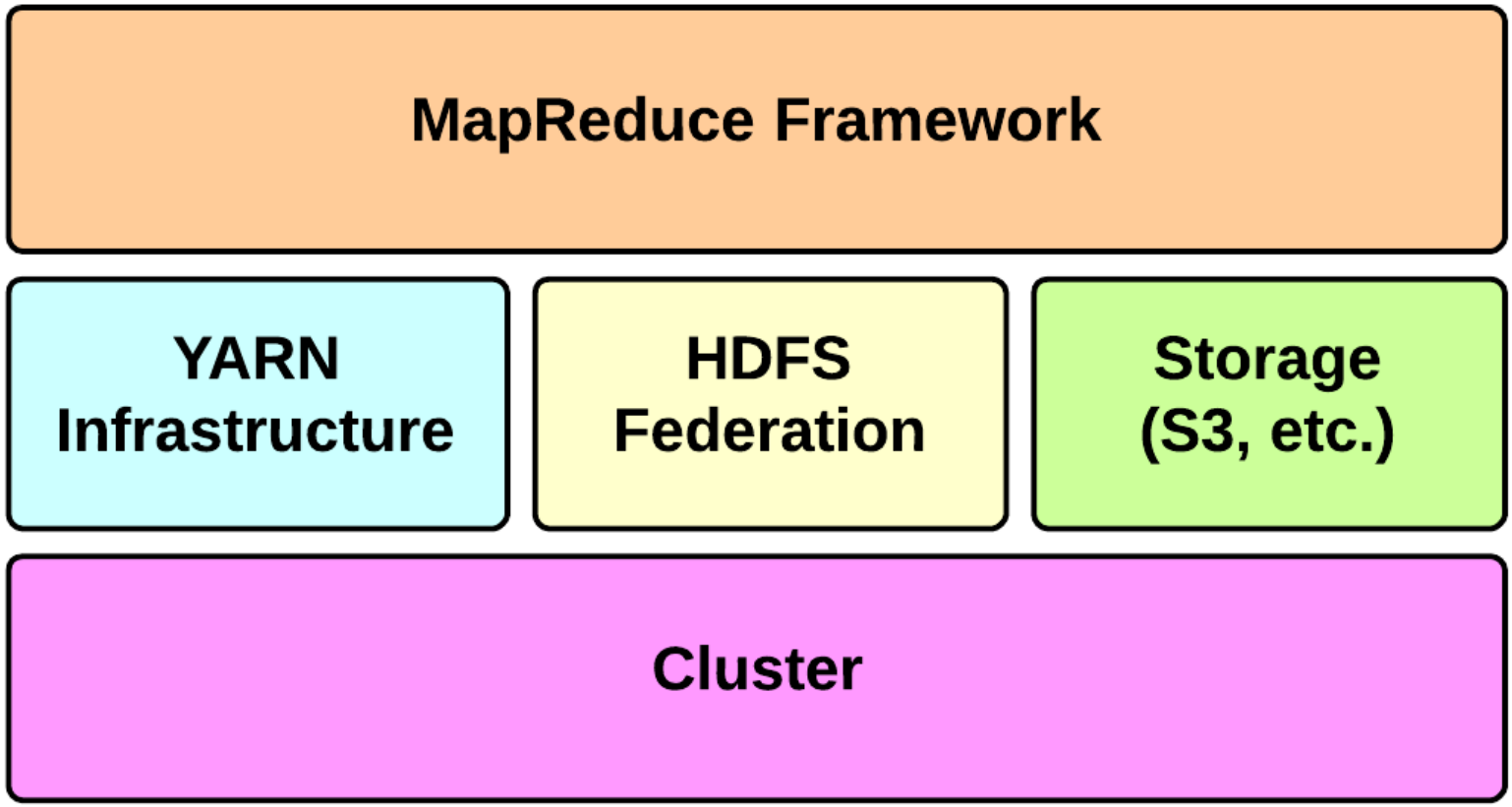
Cluster: set of host machines (**nodes**). Nodes may be partitioned in **racks**. This is the hardware part of the infrastructure.

Apache Hadoop Architecture



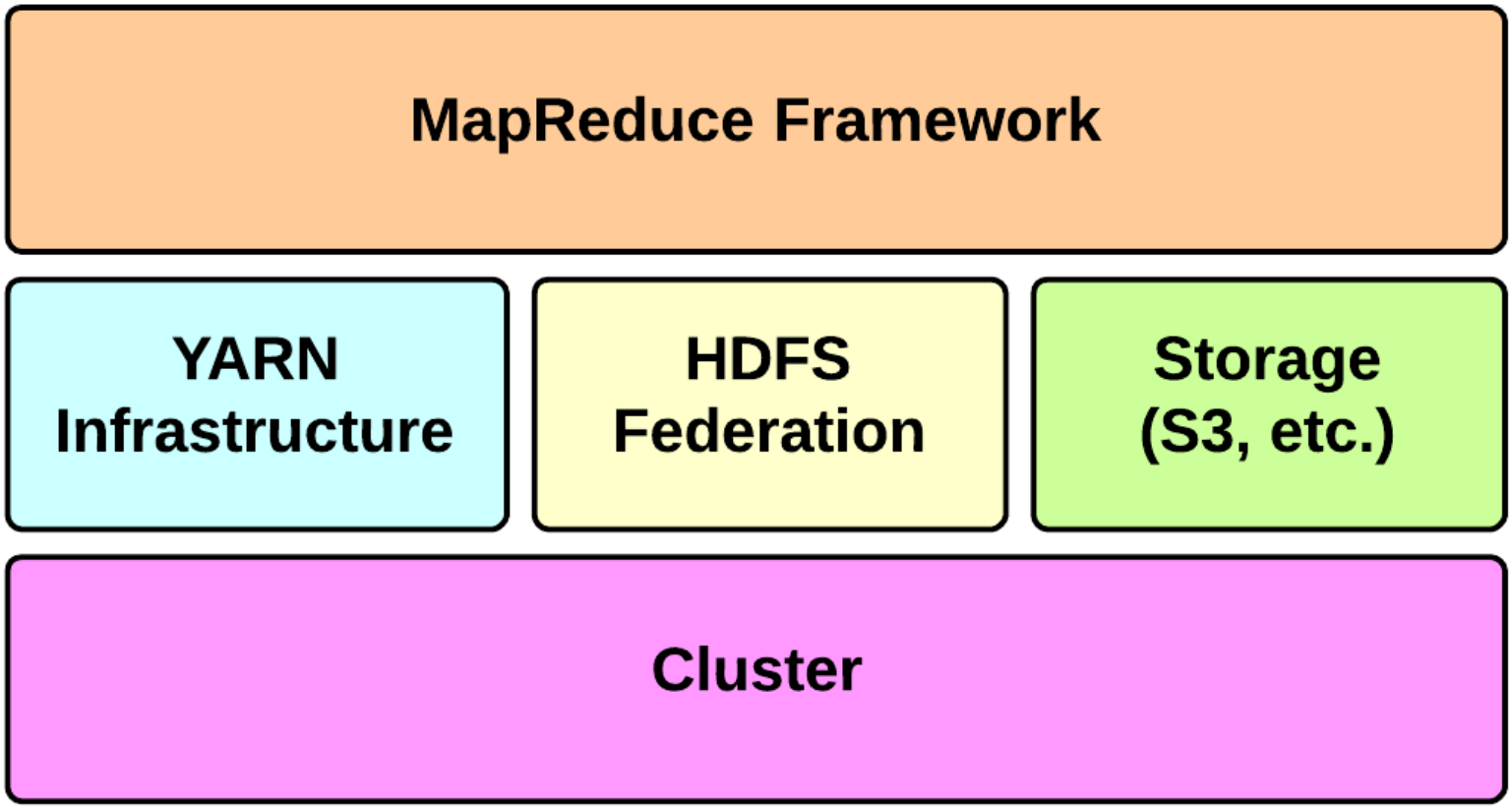
YARN: Yet Another Resource Negotiator – framework responsible for providing the computational resources (e.g., CPUs, memory, etc.) needed for application executions.

Apache Hadoop Architecture



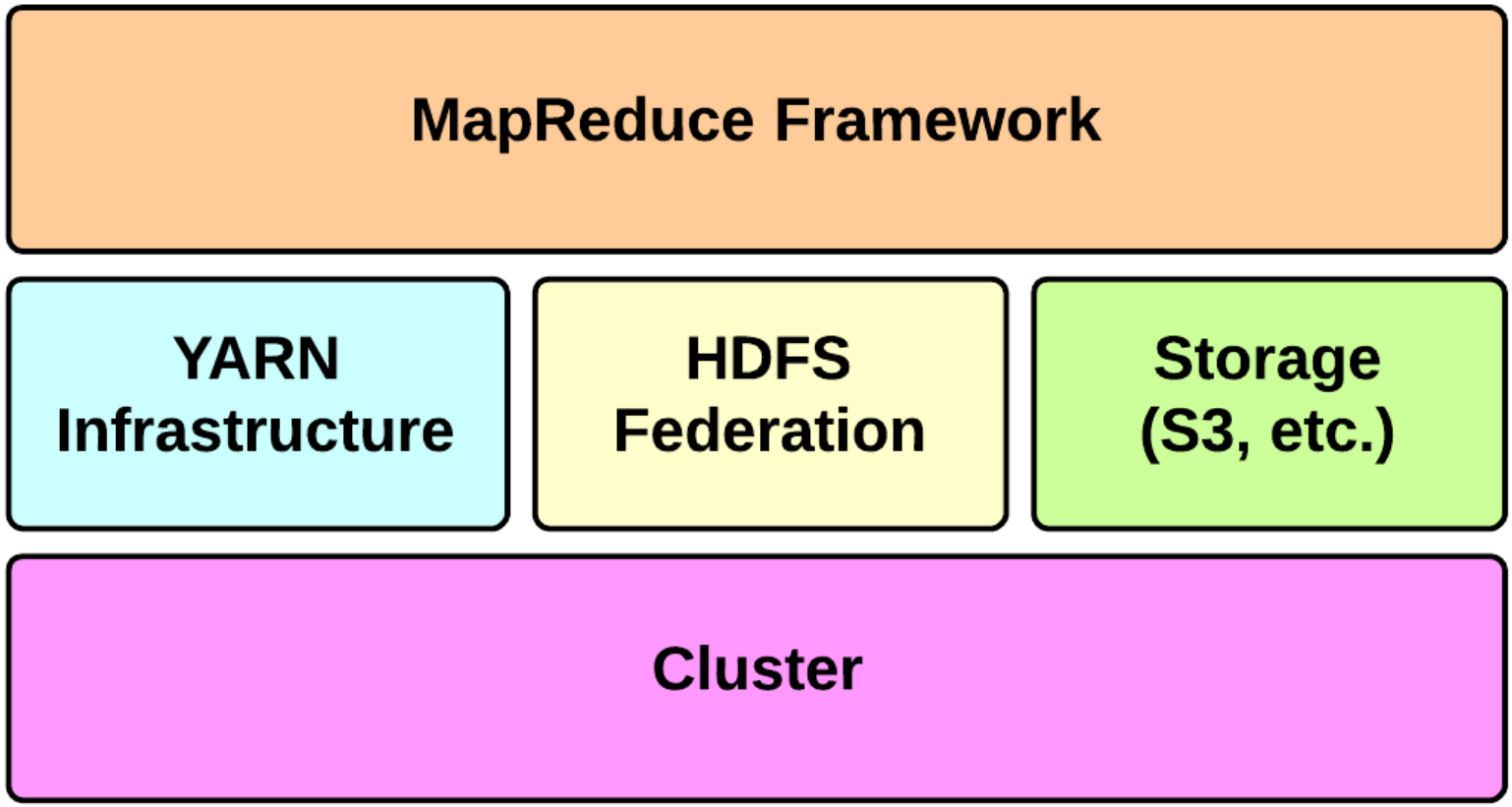
HDFS: framework responsible for providing permanent, reliable and distributed storage. This is typically used for storing inputs and output (but not intermediate ones).

Apache Hadoop Architecture



Storage: Other alternative storage solutions. Amazon uses the Simple Storage Service (S3).

Apache Hadoop Architecture

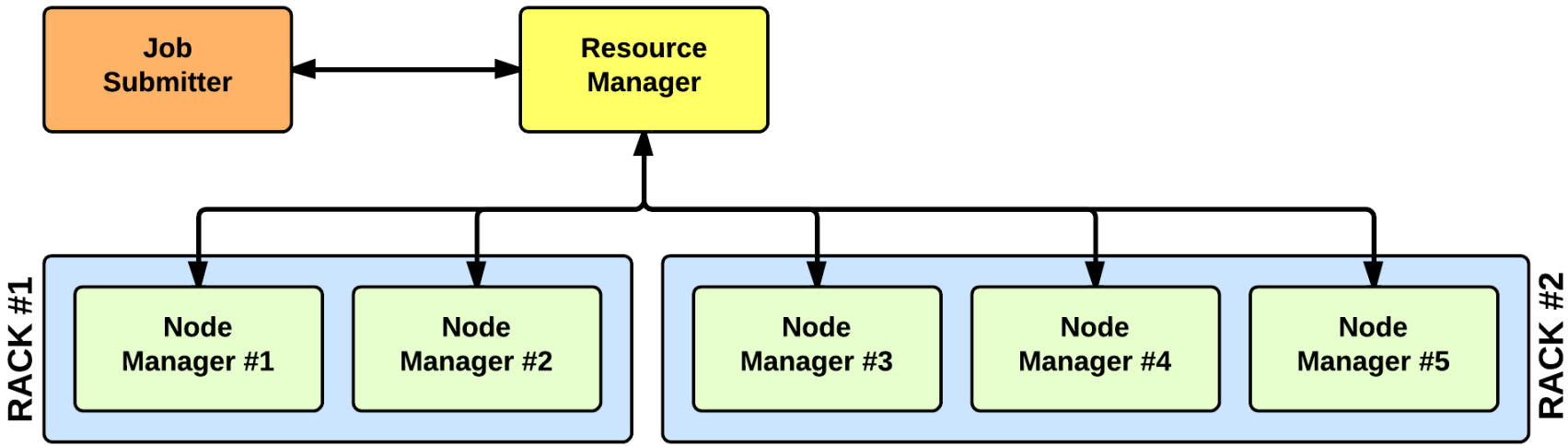


MapReduce: the software layer implementing the MapReduce paradigm. Notice that YARN and HDFS can easily support other frameworks (highly decoupled).

YARN Infrastructure:
Yet Another Resource Negotiator

YARN Infrastructure: overview

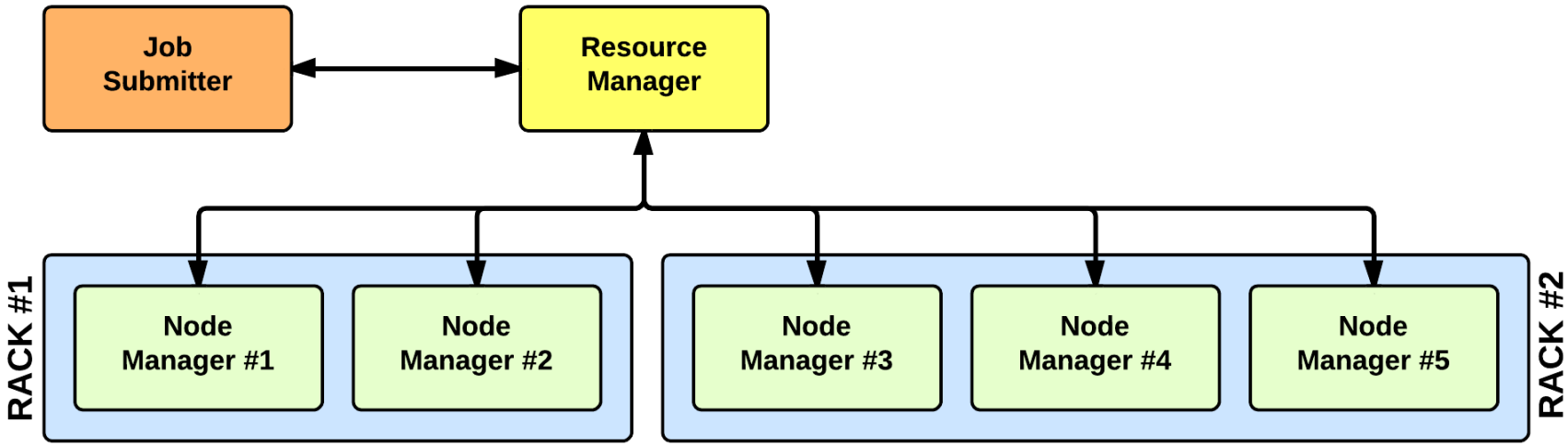
YARN handles the computational resources (CPU, memory, etc.) of the cluster. The main actors are:



YARN Infrastructure: overview

YARN handles the computational resources (CPU, memory, etc.) of the cluster. The main actors are:

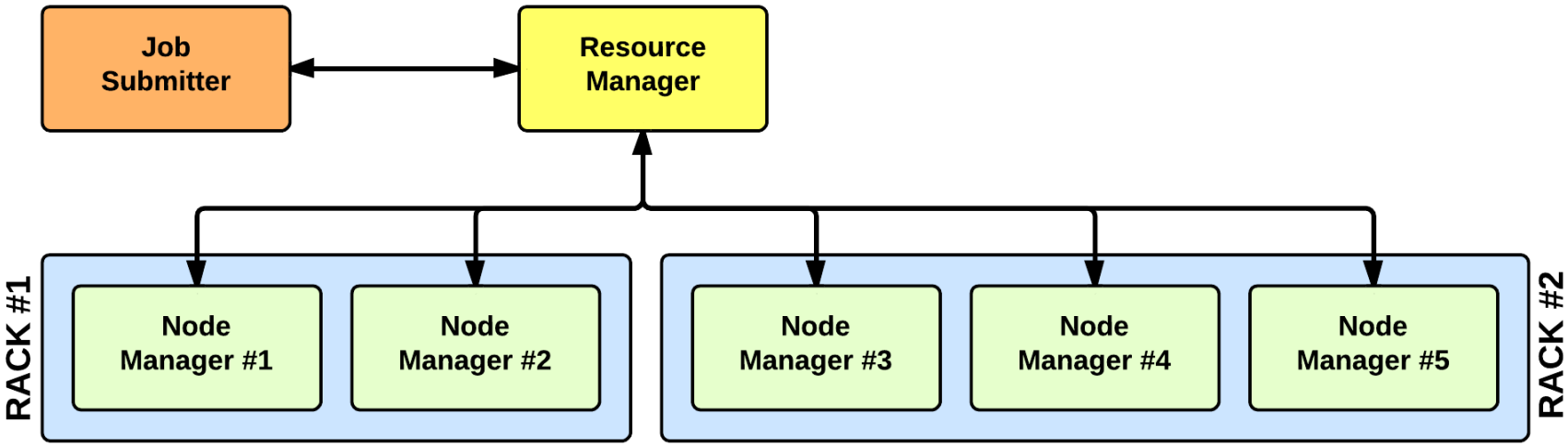
- Job Submitter: the client who submits an application



YARN Infrastructure: overview

YARN handles the computational resources (CPU, memory, etc.) of the cluster. The main actors are:

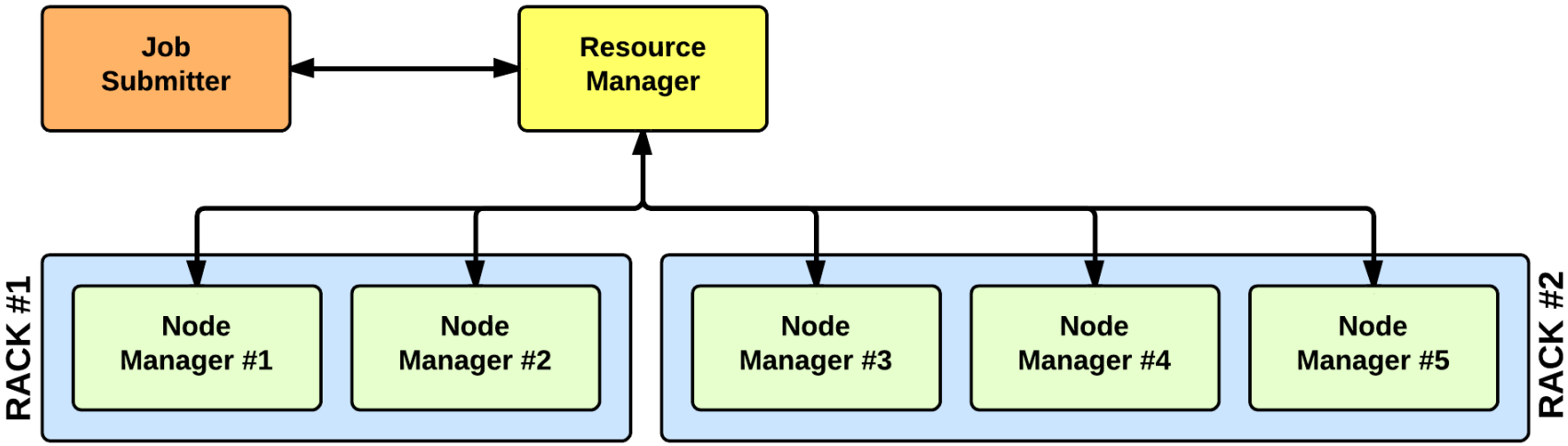
- Job Submitter: the client who submits an application
- Resource Manager: the master of the infrastructure



YARN Infrastructure: overview

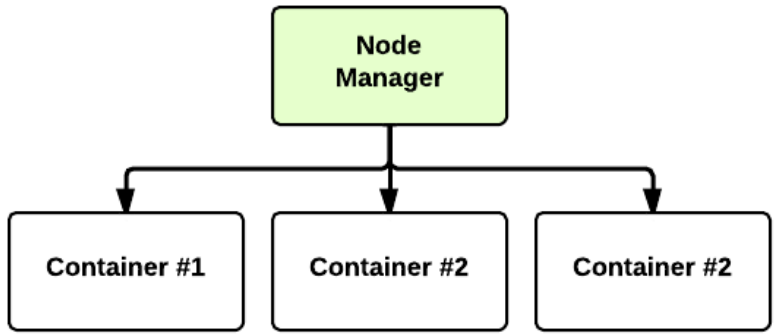
YARN handles the computational resources (CPU, memory, etc.) of the cluster. The main actors are:

- Job Submitter: the client who submits an application
- Resource Manager: the master of the infrastructure
- Node Manager: A slave of the infrastructure



YARN Infrastructure: Node Manager

The Node Manager (NM) is the slave.
When it starts, it announces himself to the RM. Periodically, it sends an heartbeat to the RM. Its resource capacity is the amount of memory and the number of vcores.

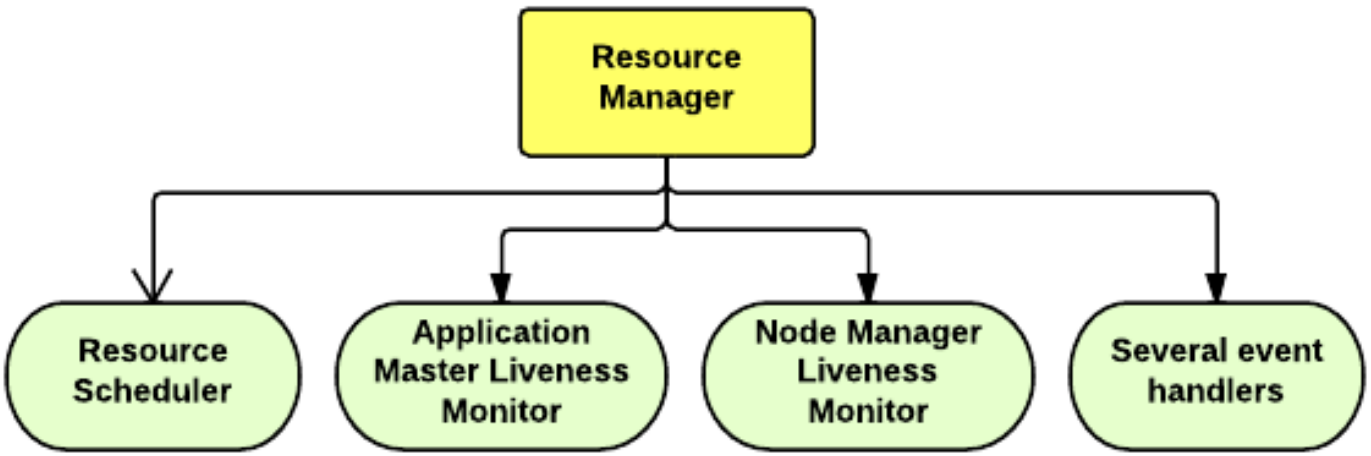


A container is a fraction of the NM capacity:

$$\begin{aligned} \text{container} &:= && \textbf{(amount of memory, \# vcores)} \\ \text{\# containers (on a NM)} &\simeq && \texttt{yarn.nodemanager.resource.memory-mb} \,/\, \texttt{yarn.scheduler.minimum-allocation-mb} \end{aligned}$$

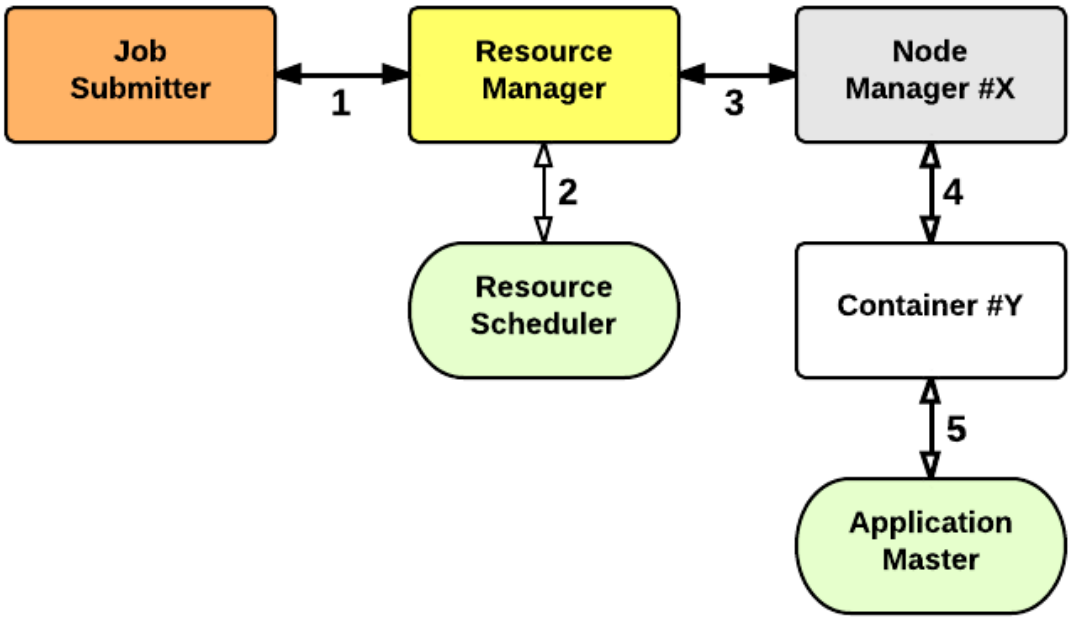
YARN Infrastructure: Resource Manager

The Resource Manager (RM) is the master. It knows where the Node Managers are located (Rack Awareness) and how many resources (containers) they have. It runs several services, the most important is the Resource Scheduler.



YARN Infrastructure: Application Startup

1. a client submits an application to the RM
2. the RM allocates a container
3. the RM contacts the NM
4. the NM launches the container
5. the container executes the **Application Master**



YARN Infrastructure: Application Master

The AM is responsible for the execution of an application. It asks for containers to the Resource Scheduler (RM) and executes specific programs (e.g., the `main` of a Java class) on the obtained containers. The AM is framework-specific.

The RM is a single point of failure in YARN. Using AMs, YARN is spreading over the cluster the metadata related to the running applications.

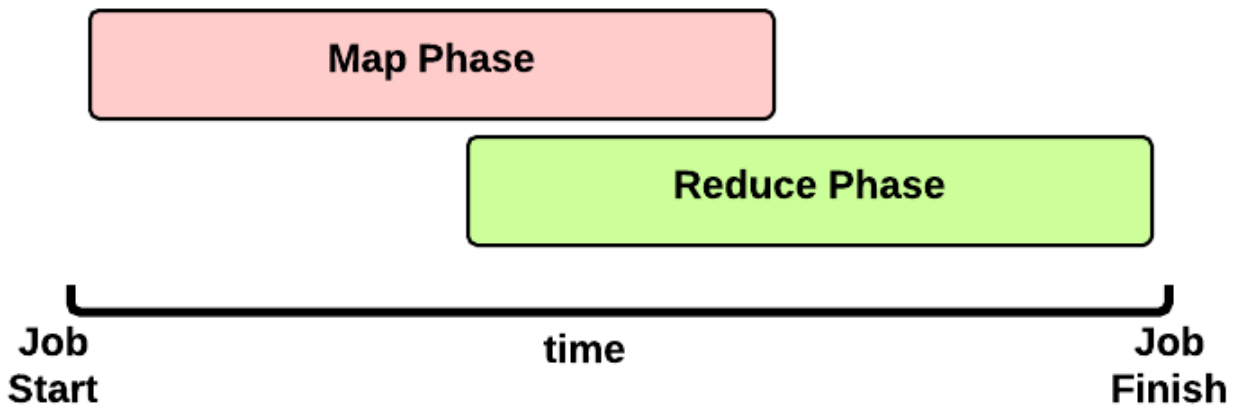
RM: reduced load & fast recovery

MapReduce Framework:
Anatomy of MR Job

MapReduce: Application $\simeq$ MR Job

Timeline of a MR Job execution:

- Map Phase: executed several **Map Tasks**
- Reduce Phase: executed several **Reduce Tasks**



The **MRAppMaster** is the director of the job.

MapReduce: what does the user give us?

A Job submitted by a user is composed by:

- a configuration: if partial then use global/default values
- a JAR containing:
  - a `map()` implementation
  - a combine implementation
  - a `reduce()` implementation
- input and output information:
  - **input directory: are they on HDFS? S3? How many files?**
  - output directory: where? HDFS? S3?

Map Phase: How many Map Tasks?

One Map Task for each input split (Job Submitter):

```
num_splits = 0
for each input file f:
    remaining = f.length
    while remaining / split_size > split_slope:
        num_splits += 1
        remaining  -= split_size
```

where:

`split_slope` $= 1.1$

`split_size` $\simeq$ **dfs.blocksize**

`mapreduce.job.maps` is ignored in MRv2 (before it was an hint)!

Map Phase: MapTask launch

The MRAppMaster immediately asks for containers needed by all MapTasks:

$$\Longrightarrow \texttt{num\_splits} \text{ container requests}$$

A container request for a MapTask tries to exploit data locality:

- a node where input split is stored
- if not, a node in same rack
- if not, any other node

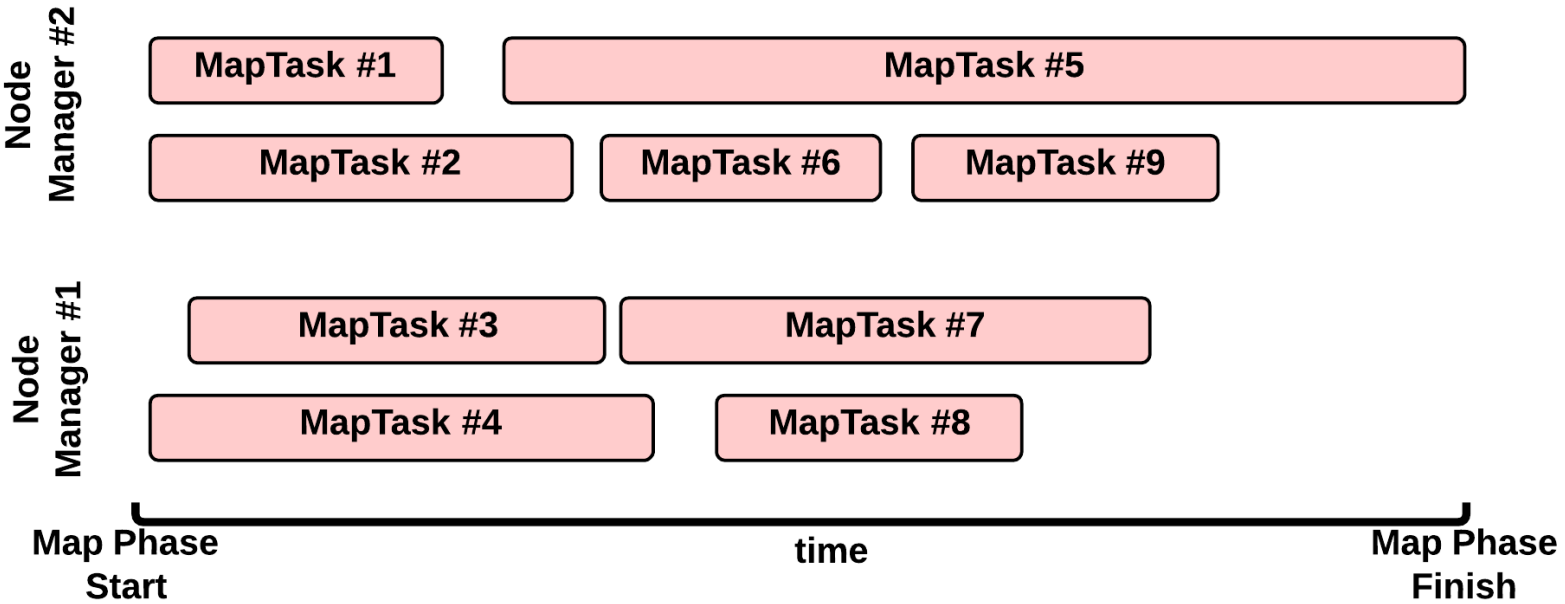
This is just an hint to the Resource Scheduler!

After a container has been assigned, the MapTask is launched.

Map Phase: Execution Overview

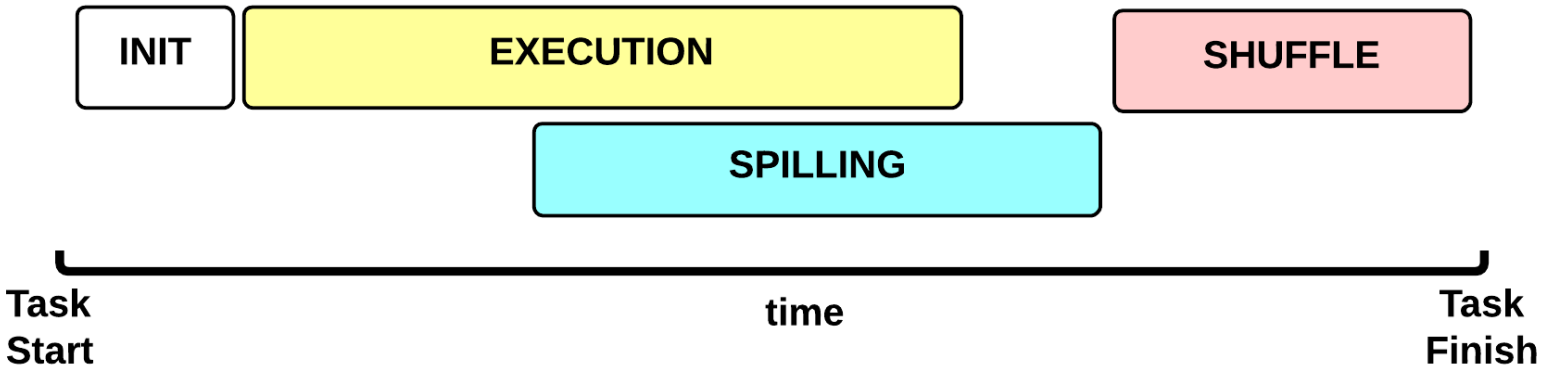
Possible execution scenario:

- 2 Node Managers (capacity $\simeq$ 2 containers)
- no other running applications
- 8 input splits



Map Phase: MapTask

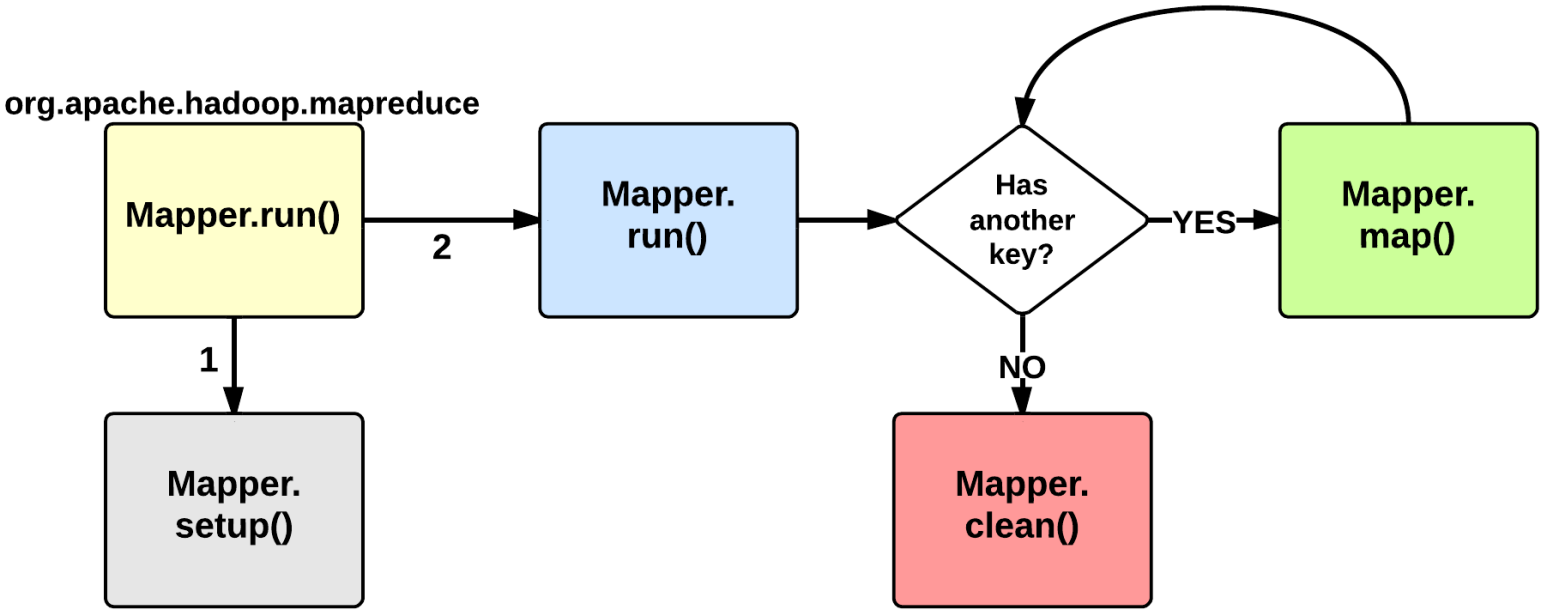
Execution timeline:



Map Phase: MapTask – Init

1. create a context (TaskAttemptContext)
2. create an instance of the user Mapper class
3. setup input (InputFormat, InputSplit, RecordReader)
4. setup output (NewOutputCollector)
5. create a mapper context (MapContext, Mapper.Context)
6. initialize input, e.g.:
   - create a SplitLineReader obj
   - create a HdfsDataInputStream obj

Map Phase: MapTask – Execution



- Mapper.Context.nextKeyValue() will load data from the input
- Mapper.Context.write() will write the output to a circular buffer

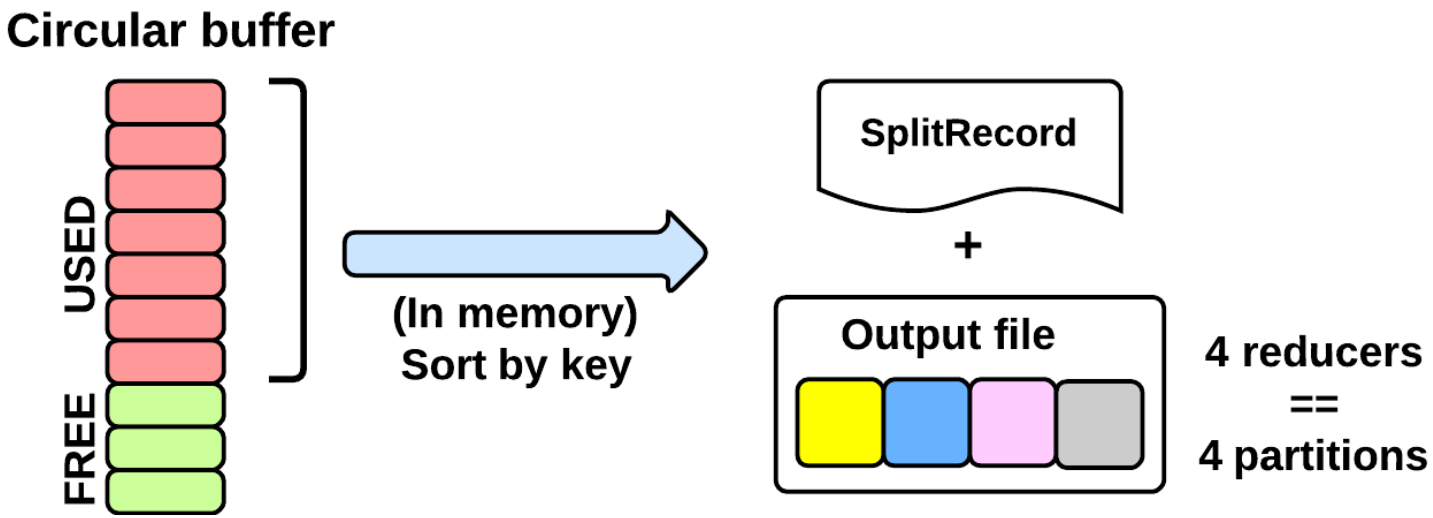
Map Phase: MapTask – Spilling

Mapper.Context.write() writes to a MapOutputBuffer of size `mapreduce.task.io.sort.mb` (100MB). If it is `mapreduce.map.sort.spill.percent` (80%) full, then **parallel** spilling phase is started.

If the circular buffer is 100% full, then `map()` is blocked!

Map Phase: MapTask – Spilling (2)

1. create a SpillRecord & create a FSOutputStream (local fs)
2. in-memory sort the chunk of the buffer (quicksort):
   - sort by <partitionIdx, key>
3. divide in partitions:
   - 1 partition for each reducer (`mapreduce.job.reduces`)
   - write partitions into output file



Map Phase: MapTask – Spilling (partitioning)

How do we partition the <key, value> tuples?

During a `Mapper.Context.write()`:

```
partitionIdx = (key.hashCode() & Integer.MAX_VALUE)
                % numReducers
```

Stored as metadata of the tuple in circular buffer.

Use `mapreduce.job.partitioner.class` for a custom partitioner

Map Phase: MapTask – Spilling (combine)

If the user specifies a combiner then, before writing the tuples to the file, we apply it on tuples of a partition:

1. create an instance of the user Reducer class
2. create a Reducer.Context: output on the local fs file
3. execute Reduce.run(): see Reduce Task slides

The combiner typically use the same implementation of the `reduce()` function and thus can be seen as a **local reducer**.

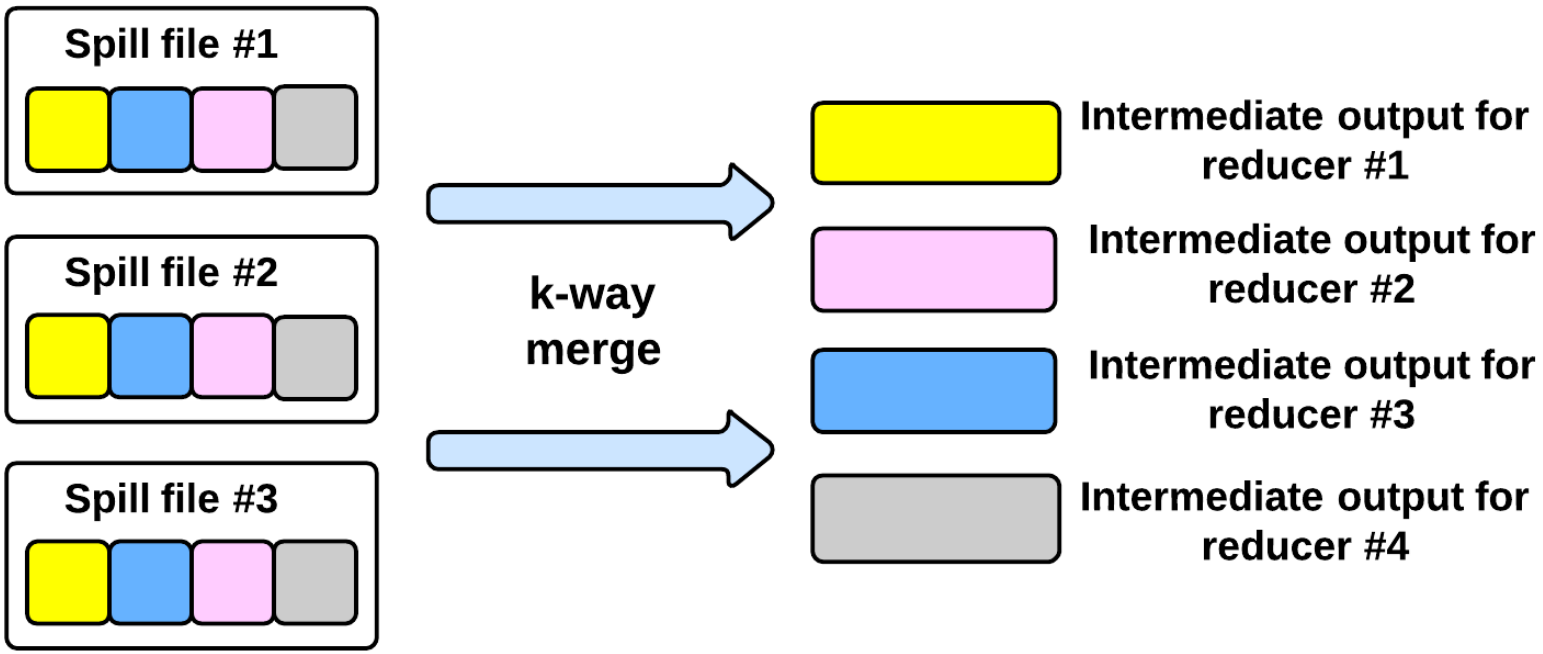
Map Phase: MapTask – Spilling (end of execution)

At the end of the execution of the Mapper.run():

1. sort and spill the remaining unspilled tuples
2. start the **shuffle** phase

Map Phase: MapTask – Shuffle

Spill files need to be merged: this is done by a k-way merge where k is equal to `mapreduce.task.io.sort.factor` (100).

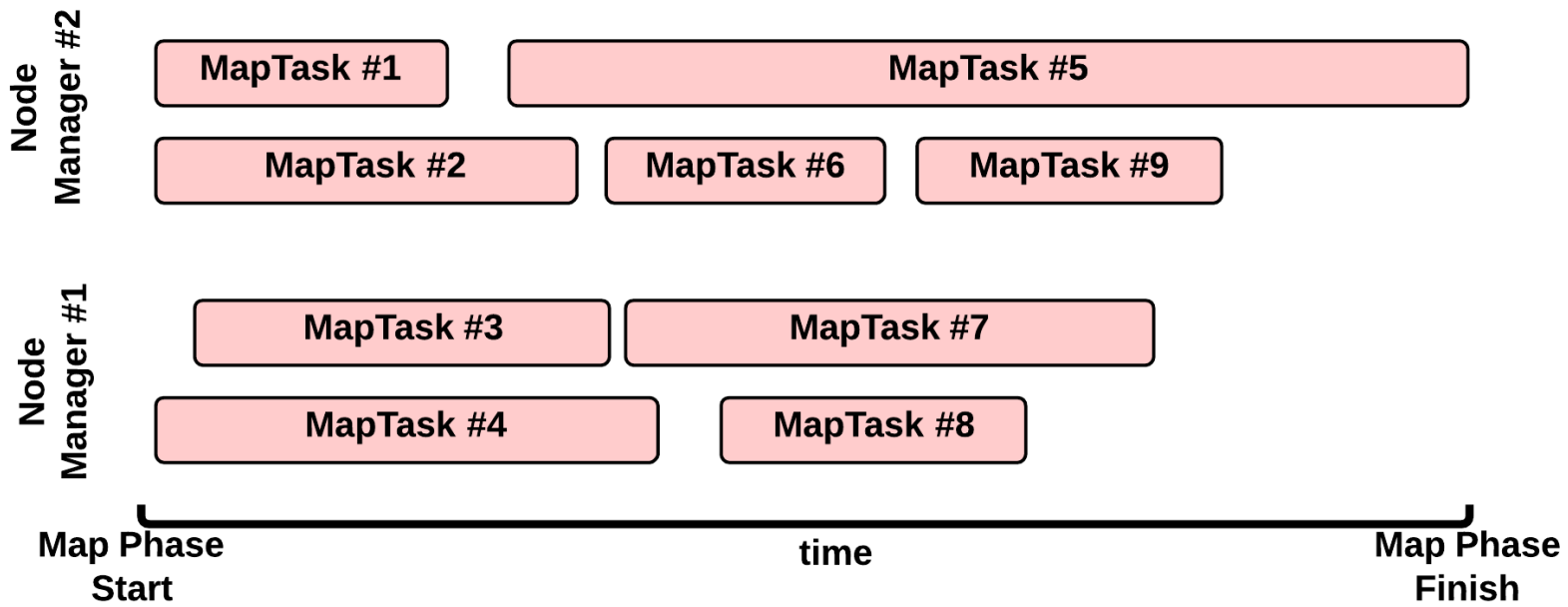


These are intermediate output files of only **one** MapTask!

Map Phase: Execution Overview

Possible execution scenario:

- 2 Node Managers (capacity $\simeq$ 2 containers)
- no other running applications
- 8 input splits



The Node Managers **locally** store the map outputs (reduce inputs).

Reduce Phase: Reduce Task Launch

The **MRAppMaster** waits until `mapreduce.job.reduce.slowstart.completedmaps` (5%) MapTasks are completed.
Then (periodically executed):

- if all maps have a container assigned then all (remaining) reducers are scheduled
- otherwise it checks percentage of completed maps:
  - check available cluster resources for the app
  - check resource needed for unassigned rescheduled maps
  - ramp down (unschedule/kill) or ramp up (schedule) reduce tasks

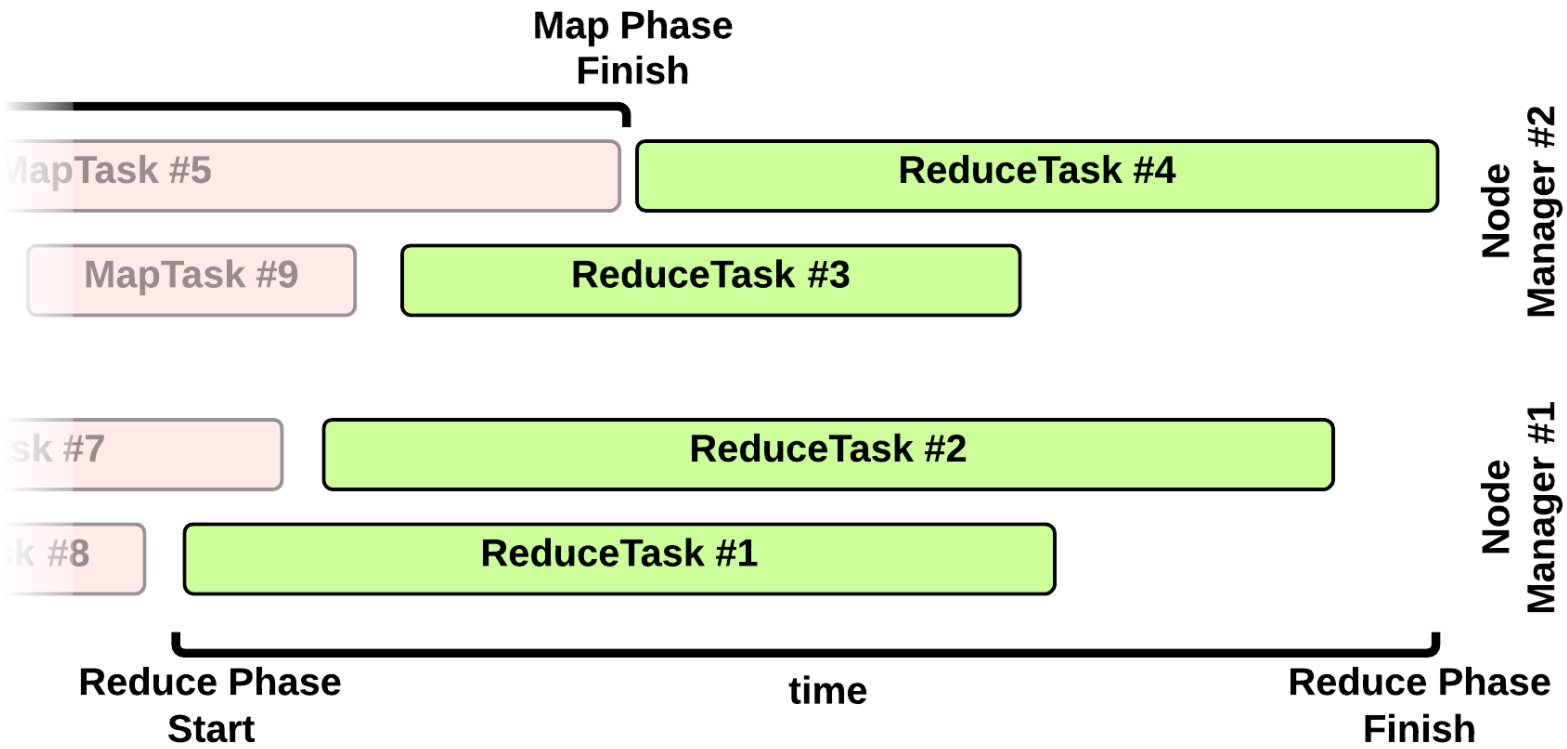
When a reduce task is scheduled, a container request is made. This does NOT exploit data locality.

A MapTask request has a higher priority than Reduce Task request.

Reduce Phase: Execution Overview

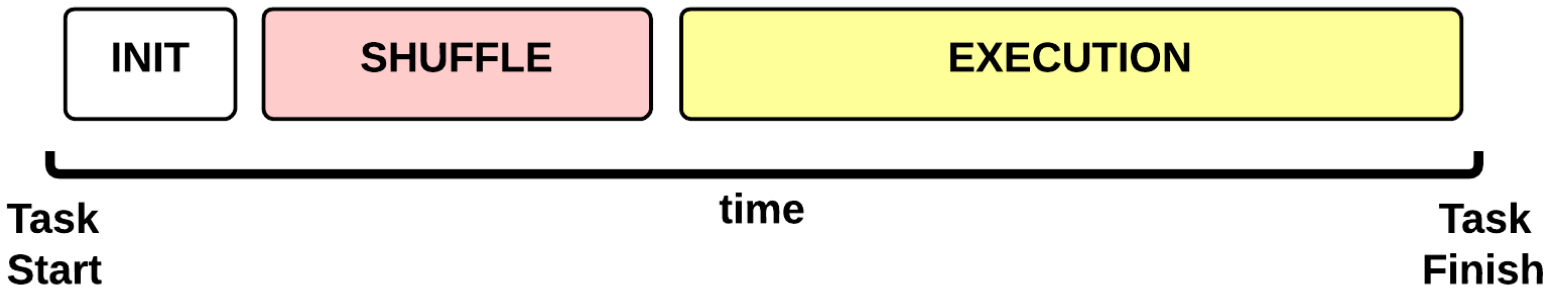
Possible execution scenario:

- 2 Node Managers (capacity $\simeq$ 2 containers each)
- no other running applications
- 4 reducers (`mapreduce.job.reduces`, default: 1)



Reduce Phase: Reduce Task

Execution timeline:



Reduce Phase: Reduce Task – Init

1. init a codec (if map outputs are compressed)
2. create an instance of the combine output collector (if needed)
3. create an instance of the shuffle plugin (`mapreduce.job.reduce.shuffle.consumer.plugin.class`, default: org.apache.hadoop.mapreduce.task.reduce.Shuffle.class)
4. create a shuffle context (ShuffleConsumerPlugin.Context)

Reduce Phase: Reduce Task – Shuffle

The shuffle has two steps:

1. fetch map outputs from Node Managers
2. merge them

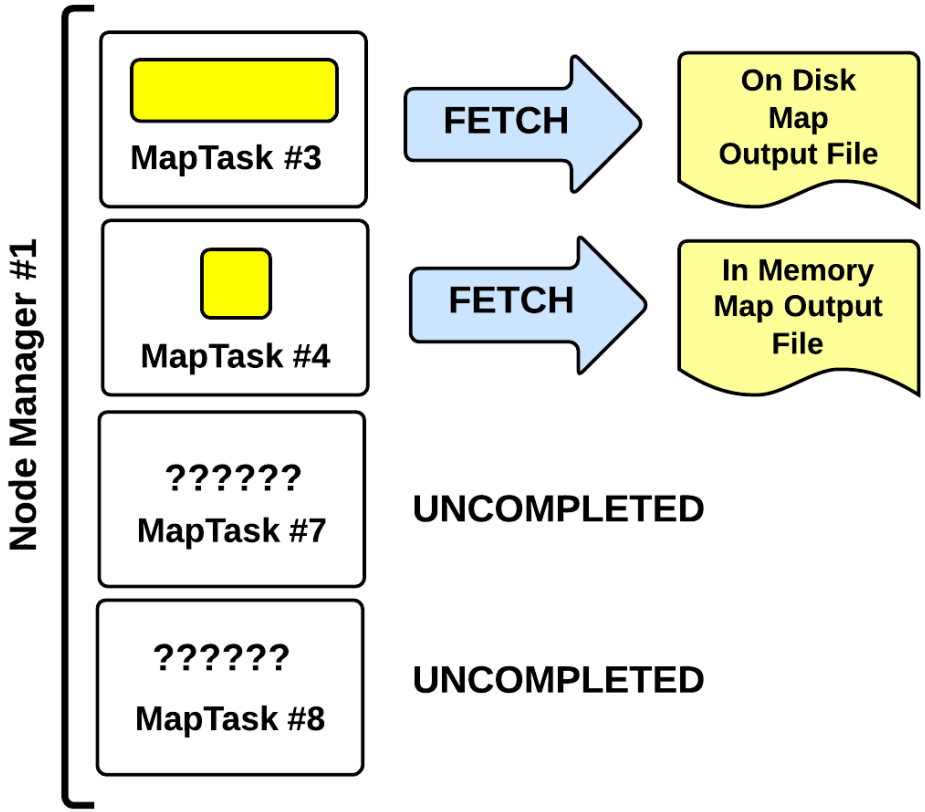
Reduce Phase: Reduce Task – Shuffle (fetch)

Several parallel fetchers are started (up to `mapreduce.reduce.shuffle.parallelcopies`, default: 5). Each fetcher collects map outputs from one NM (possibly many containers):

- if output size less than 25% of NM memory then create an in memory output (wait until enough memory is available)
- otherwise create a disk output

Reduce Phase: Reduce Task – Shuffle (fetch) (2)

Fetch the outputs over HTTP and add to related merge queue.

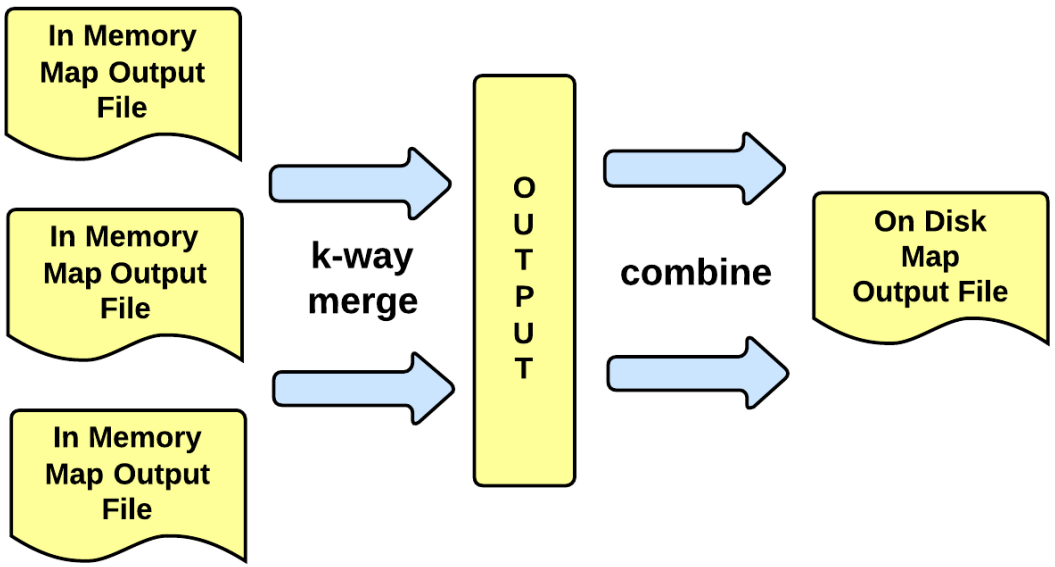


A Reduce Task may start before the end of the Map Phase thus you can fetch only from completed map tasks. Periodically repeat fetch process.

Reduce Phase: Reduce Task – Shuffle (in memory merge)

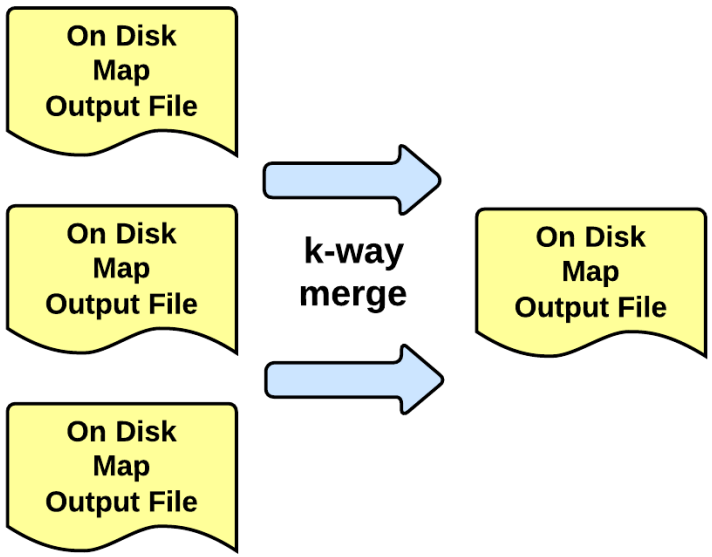
The in memory merger:

1. perform a k-way merge
2. run the combiner (if needed)
3. result is written on a On Disk Map Output and it is queued



Reduce Phase: Reduce Task – Shuffle (on disk merge)

Extract from the queue, k-way merge and queue the result:

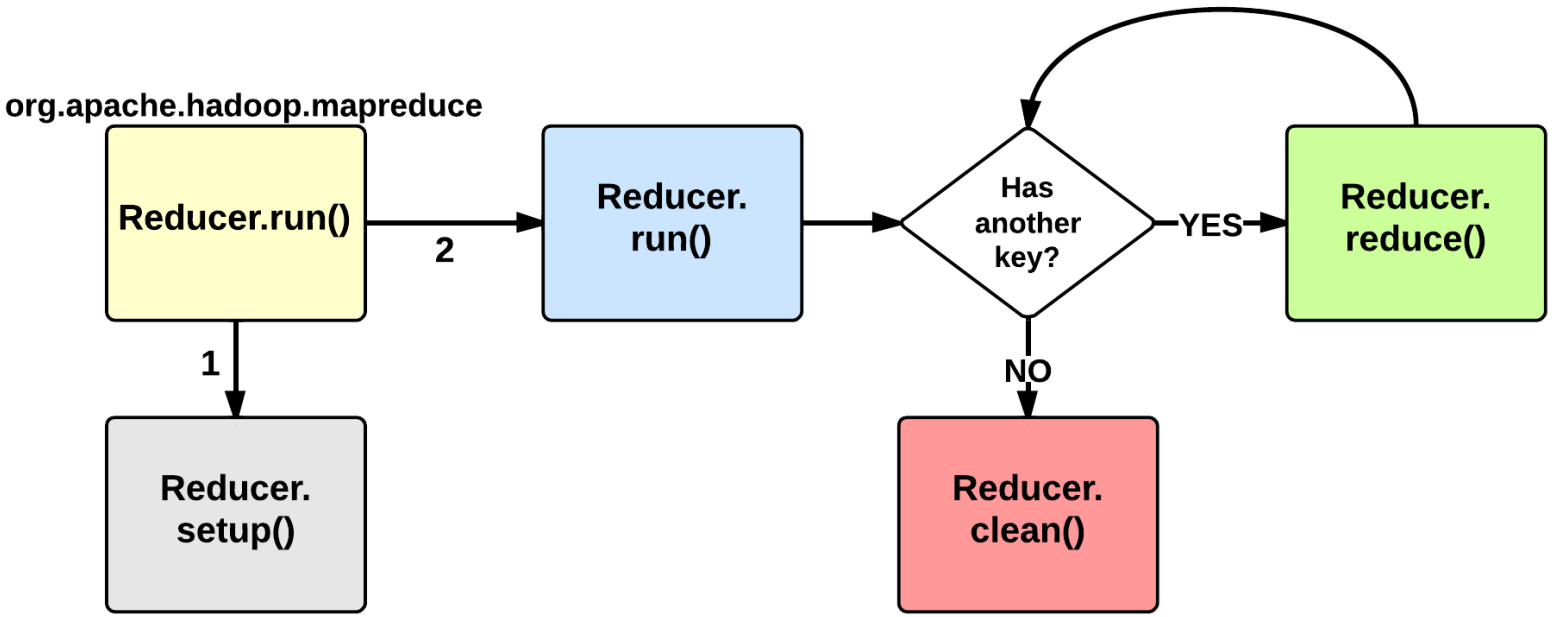


Stop when all files has been merged together: the final merge will provide a RawKeyValueIterator instance (input of the reducer).

Reduce Phase: Reduce Task – Execution (init)

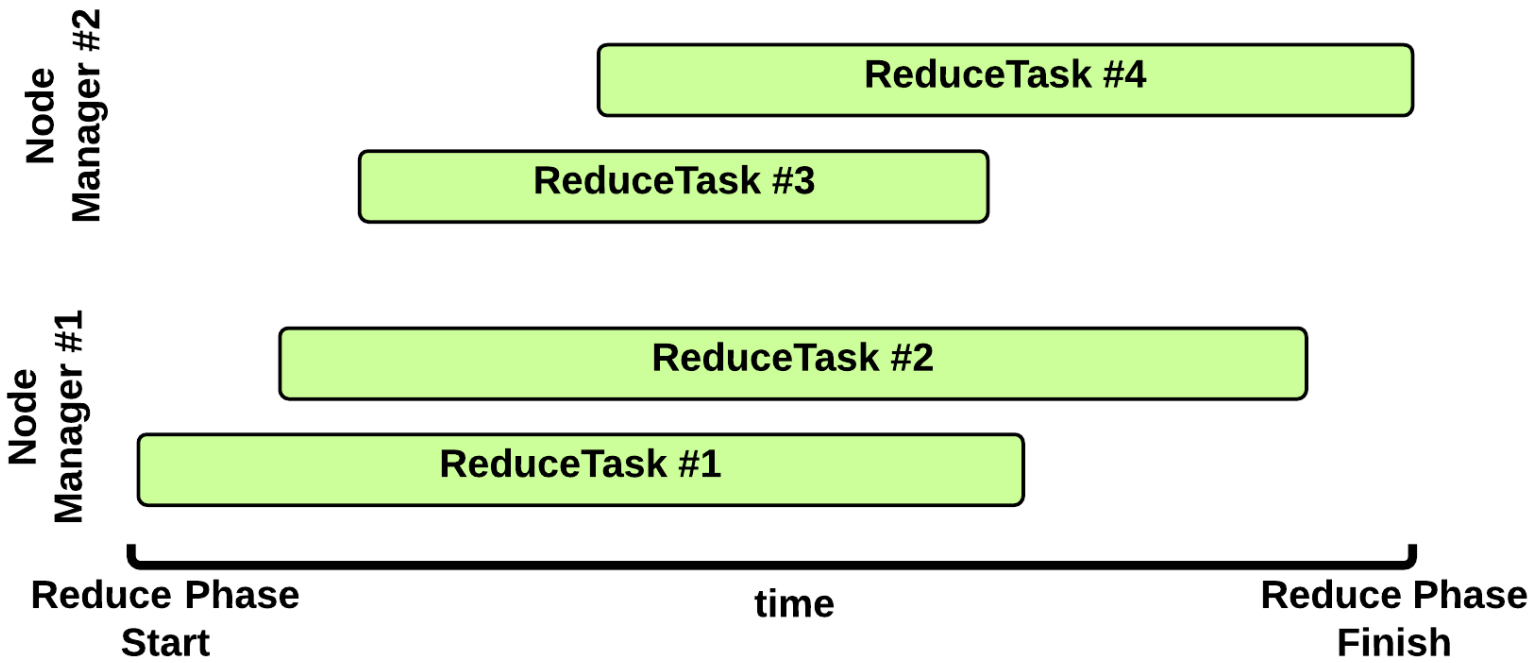
1. create a context (TaskAttemptContext)
2. create an instance of the user Reduce class
3. setup output (RecordWriter, TextOutputFormat)
4. create a reducer context (Reducer.Context)

Reduce Phase: Reduce Task – Execution (run)



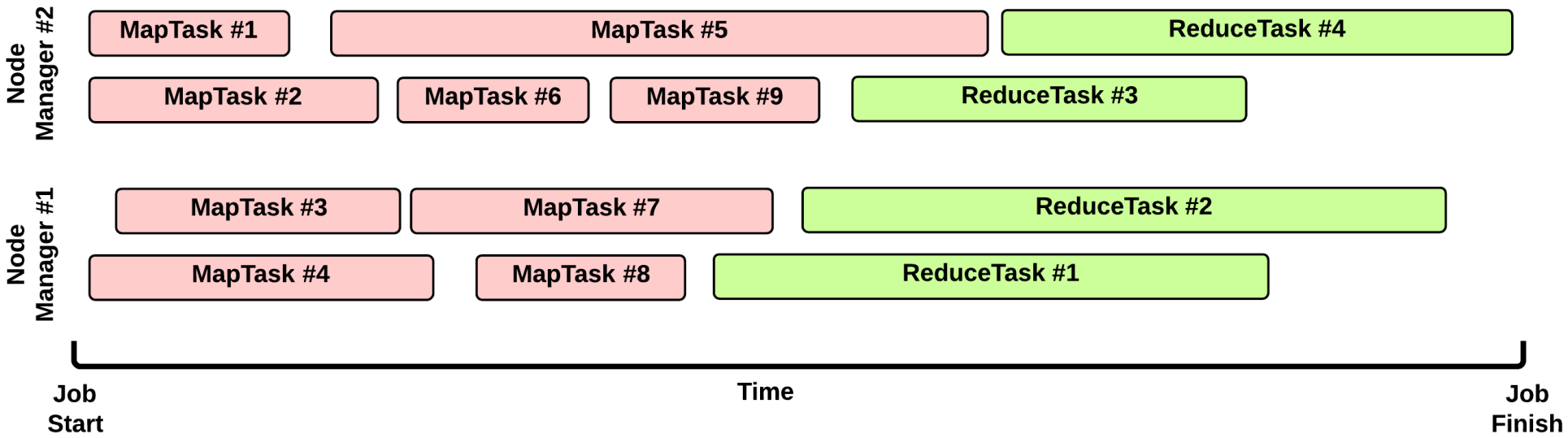
The output is typically written on HDFS file.

Reduce Phase: Execution Overview



MapReduce: Application ≃ MR Job

Possible execution timeline:



That's it!

MapReduce: Task Progress

- A MapTask has two phases:
  - Map (66%): progress due to perc. of processed input
  - Sort (33%): 1 subphase for each reducer
    - subphase progress due to perc. of merged bytes
- A ReduceTask has three phases:
  - Copy (33%): progress due to perc. of fetched input
  - Sort (33%): progress due to processed bytes in final merge
  - Reduce (33%): progress due to perc. of processed input

MapReduce: Speculation

MRAppMaster may launch speculative tasks:

```
est = (ts - start) / MAX(0.0001, Status.progress())
estEndTime = start + est
estReplacementEndTime = now() + TaskDurations.mean()

if estEndTime < now() then
  return PROGRESS_IS_GOOD
elif estReplacementEndTime >= estEndTime then
  return TOO_LATE_TO_SPECULATE
else then
  return estEndTime - estReplacementEndTime // score
```

Speculate the task with highest score.

MapReduce: Application Status

The status of a MR job is tracked by the MRAppMaster using several Finite State Machines:

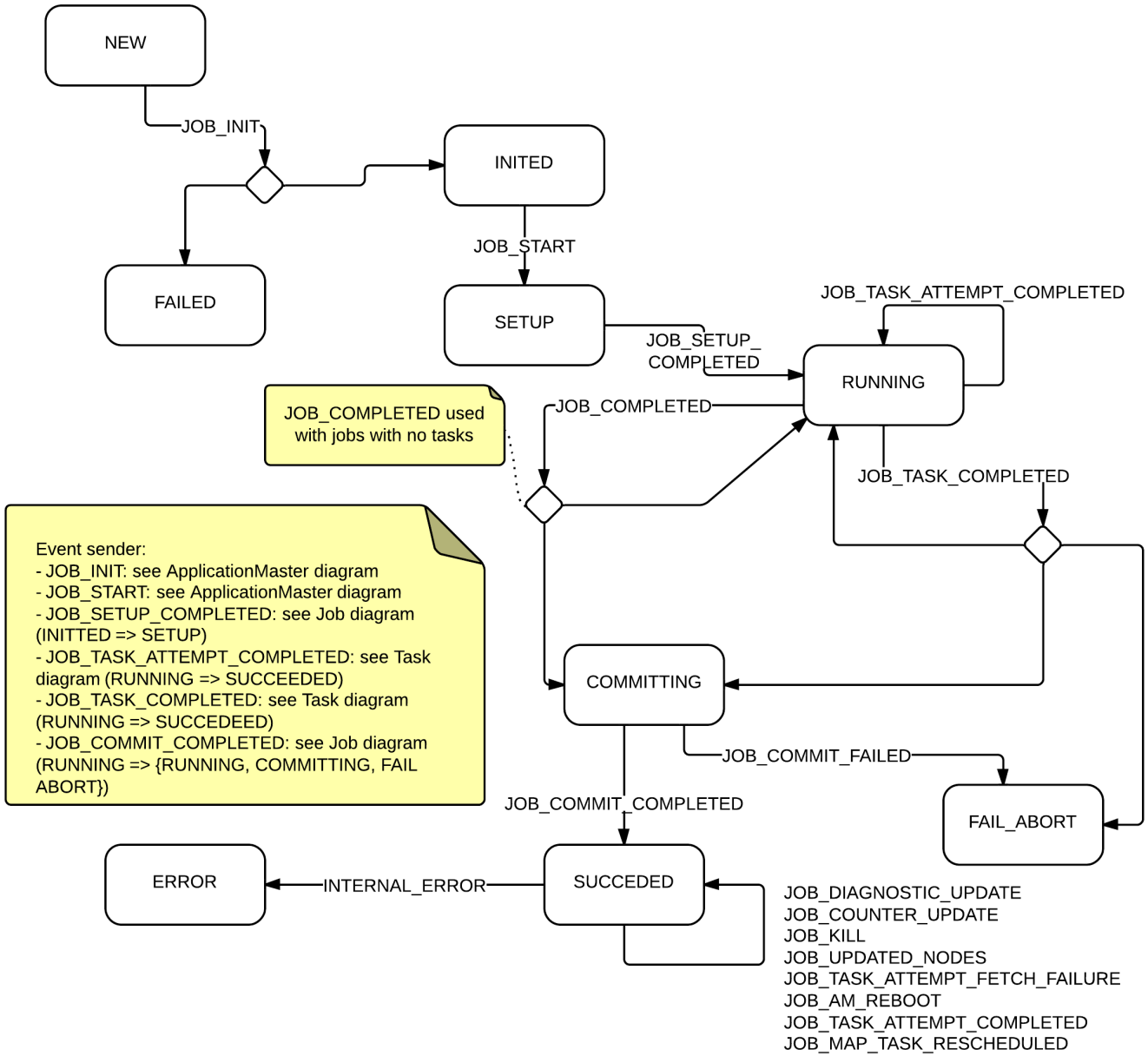
- Job: 14 states, 80 transitions, 19 events
- Task: 14 states, 36 transitions, 9 events
- Task Attempt: 13 states, 60 transitions, 17 events

A job is composed by several tasks. Each tasks may have several task attempts. Each task attempt is executed on a container.

Instead, a Node Manager maintains the states of:

- Application: 7 states, 21 transitions, 9 events
- Container: 11 states, 46 transitions, 12 events

MapReduce: Job FSM (example)



Configuration Parameters (recap)

Parameter	Meaning
`mapreduce.framework.name`	The runtime framework for executing MapReduce jobs. Set to YARN.
`mapreduce.job.reduces`	Number of reduce tasks. Default: 1
`dfs.blocksize`	HDFS block size. Default 128MB.
`yarn.resourcemanager.scheduler.class`	Scheduler class. Default: CapacityScheduler
`yarn.nodemanager.resource.memory-mb`	Memory available on a NM for containers. Default: 8192
`yarn.scheduler.minimum-allocation-mb`	Min allocation for every container request. Default: 1024
`mapreduce.map.memory.mb`	Memory request for a MapTask. Default: 1024
`mapreduce.reduce.memory.mb`	Memory request for a ReduceTask. Default: 1024

Configuration Parameters (recap) (2)

Parameter	Meaning
`mapreduce.task.io.sort.mb`	Size of the circular buffer (map output). Default: 100MB
`mapreduce.map.sort.spill.percent`	Circular buffer soft limit. Once reached, start the spilling process. Default: 0.80
`mapreduce.job.partitioner.class`	The Partitioner class. Default: HashPartitioner.class
`map.sort.class`	The sort class for sorting keys. Default: org.apache.hadoop.util.QuickSort
`mapreduce.reduce.shuffle.memory.limit.percent`	Maximum percentage of the in-memory limit that a single shuffle can consume. Default: 0.25
`mapreduce.reduce.shuffle.input.buffer.percent`	The % of memory to be allocated from the maximum heap size to storing map outputs during the shuffle. Default: 0.70

Configuration Parameters (recap) (3)

Parameter	Meaning
`mapreduce.reduce.shuffle .merge.percent`	The usage % at which an in-memory merge will be initiated. Default: 0.66
`mapreduce.map. combine.minspills`	Apply combine only if you at least this number of spill files. Default: 3.
`mapreduce.task. io.sort.factor`	The number of streams to merge at once while sorting files. Default: 100 (10)
`mapreduce.job.reduce. slowstart.completedmaps`	Fraction of the number of maps in the job which should be complete before reduces are scheduled for the job. Default: 0.05
`mapreduce.reduce. shuffle.parallelcopies`	Number of parallel transfers run by reduce during the shuffle (fetch) phase. Default: 5
`mapreduce.reduce. memory.totalbytes`	Memory of a NM. Default: Runtime.maxMemory()

Hadoop: a bad angel

Writing a MapReduce program is relatively easy. On the other hand, writing an **efficient** MapReduce program is hard:

- many configuration parameters:
  - YARN: 115 parameters
  - MapReduce: 195 parameters
  - HDFS: 173 parameters
  - core: 145 parameters
- lack of control over the execution: how to debug?
- many implementation details: what is happening?

How can we help the user?

How can we help the user?

We need profilers!

How can we help the user?

We need profilers!

My current research is focused on this goal.