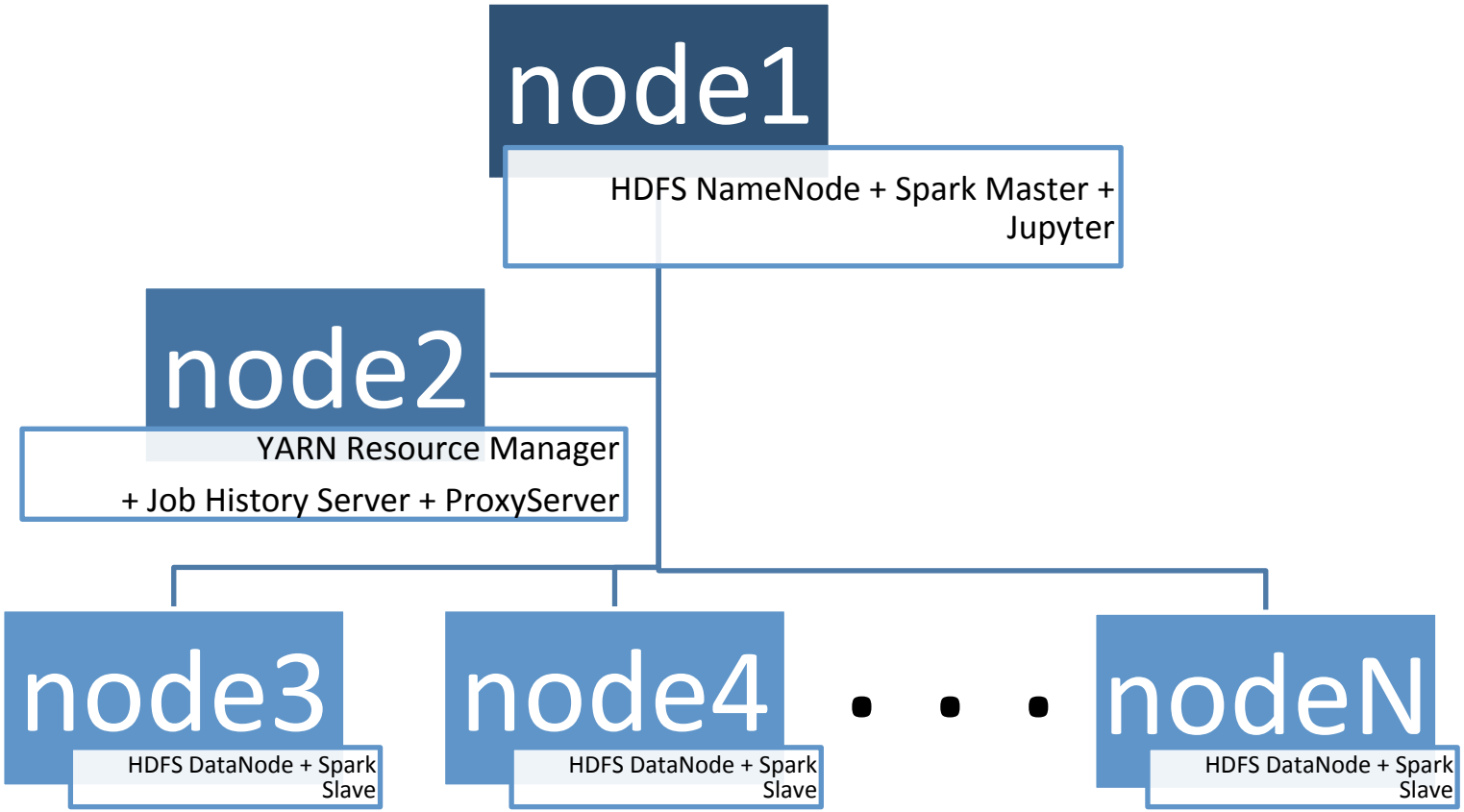
Spark Cluster Overview

Rosa Filgueira

Spark cluster configuration

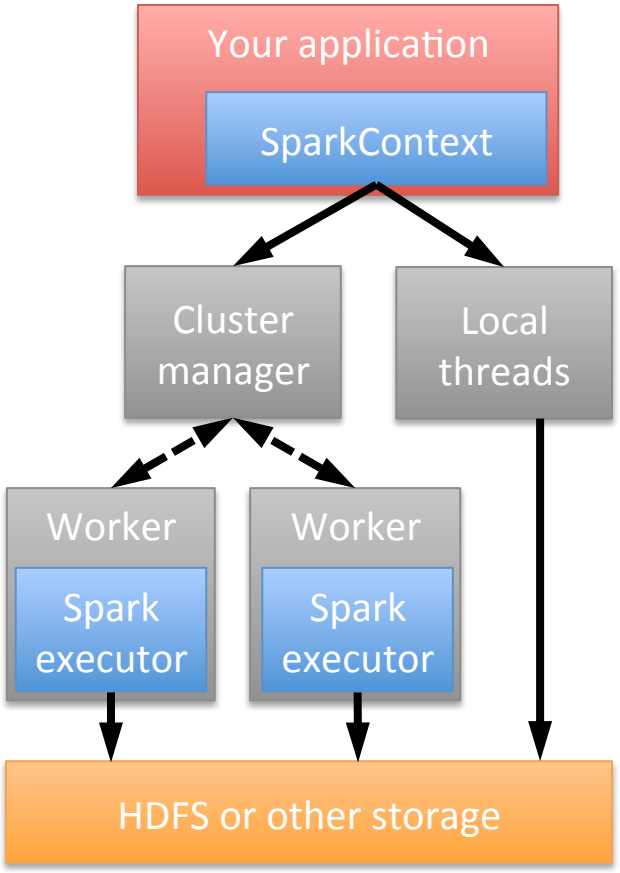


Spark cluster configuration

- Software:
  - Apache Spark-2.1.1
  - Hadoop-2.8
  - Anaconda3-4.4
  - 64-bit CentOS6.5
- 4 VMs with 2GB Memory each

Software Components

- Spark runs as a library in your program (one instance per app)
- Runs tasks locally or on a cluster
  - Standalone deploy cluster, YARN
- Accesses storage via Hadoop InputFormat API
  - Can use HBase, HDFS, S3, …



Monitoring the cluster services

- HDFS NameNode

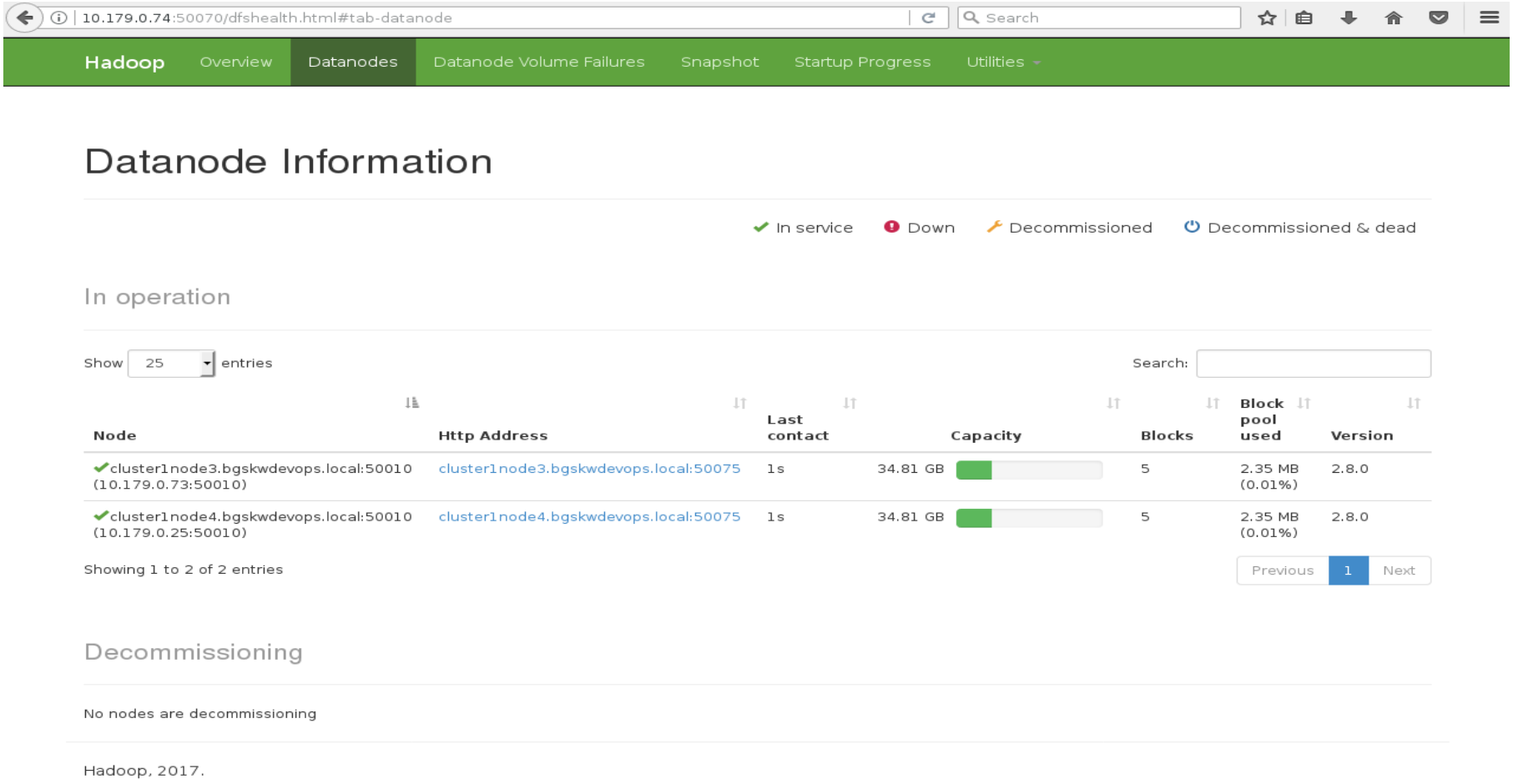
  http://10.211.55.101:50070/dfshealth.html
- Resource Manager

  http://10.211.55.102:8088/cluster
- Spark History

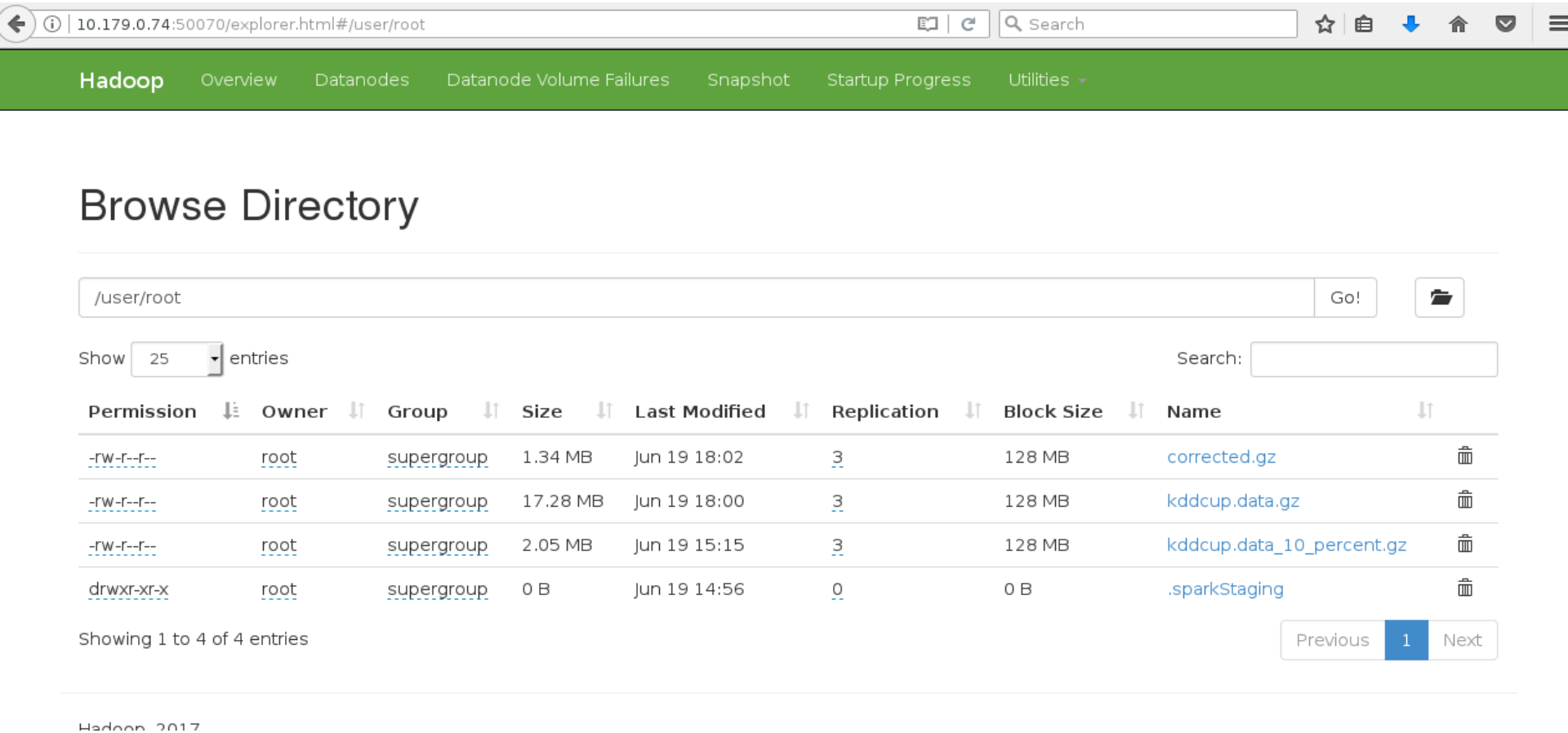
  http://10.211.55.101:18080
- Jupyter Notebook

  http://10.211.55.101:8888

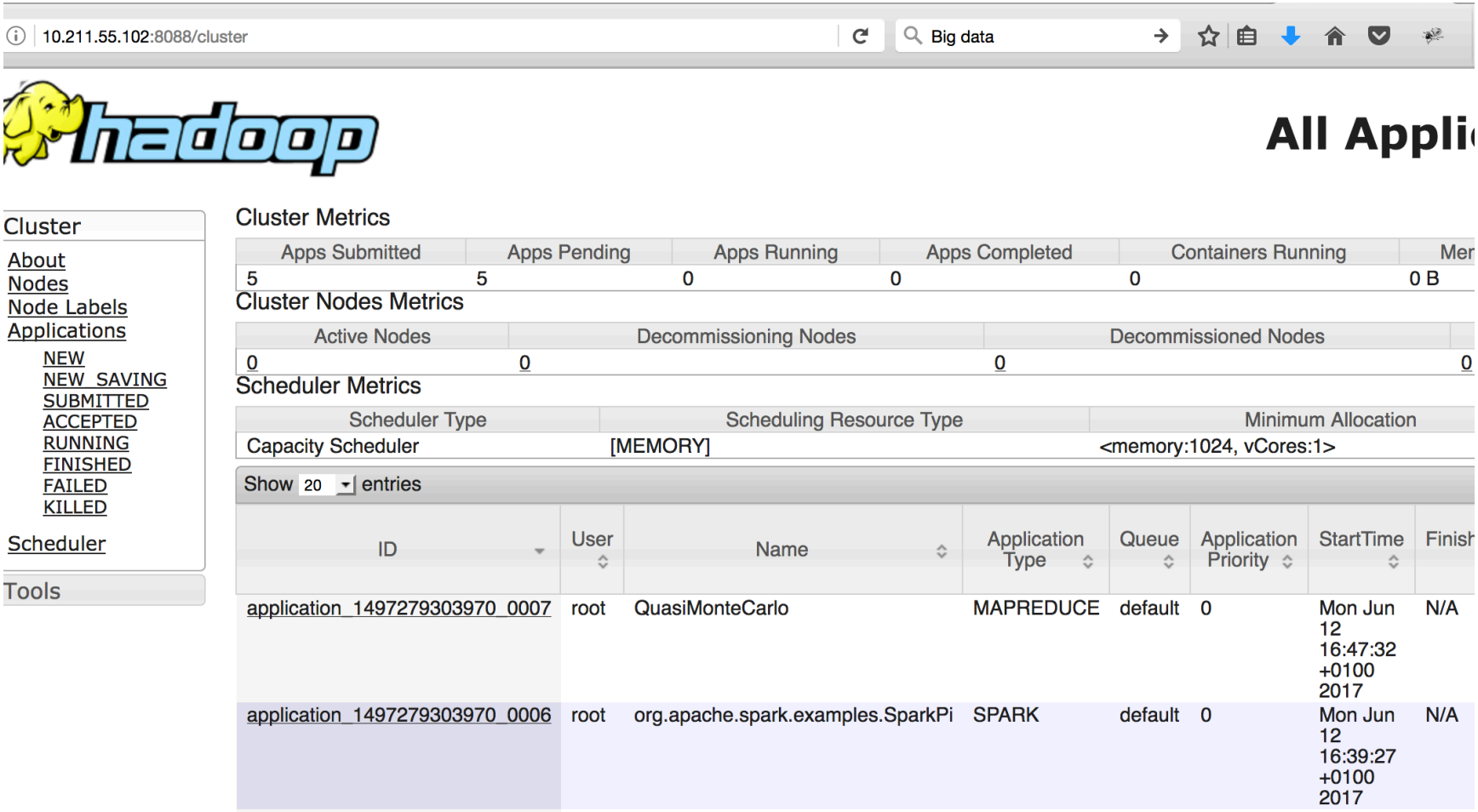
HDFS NameNode UI



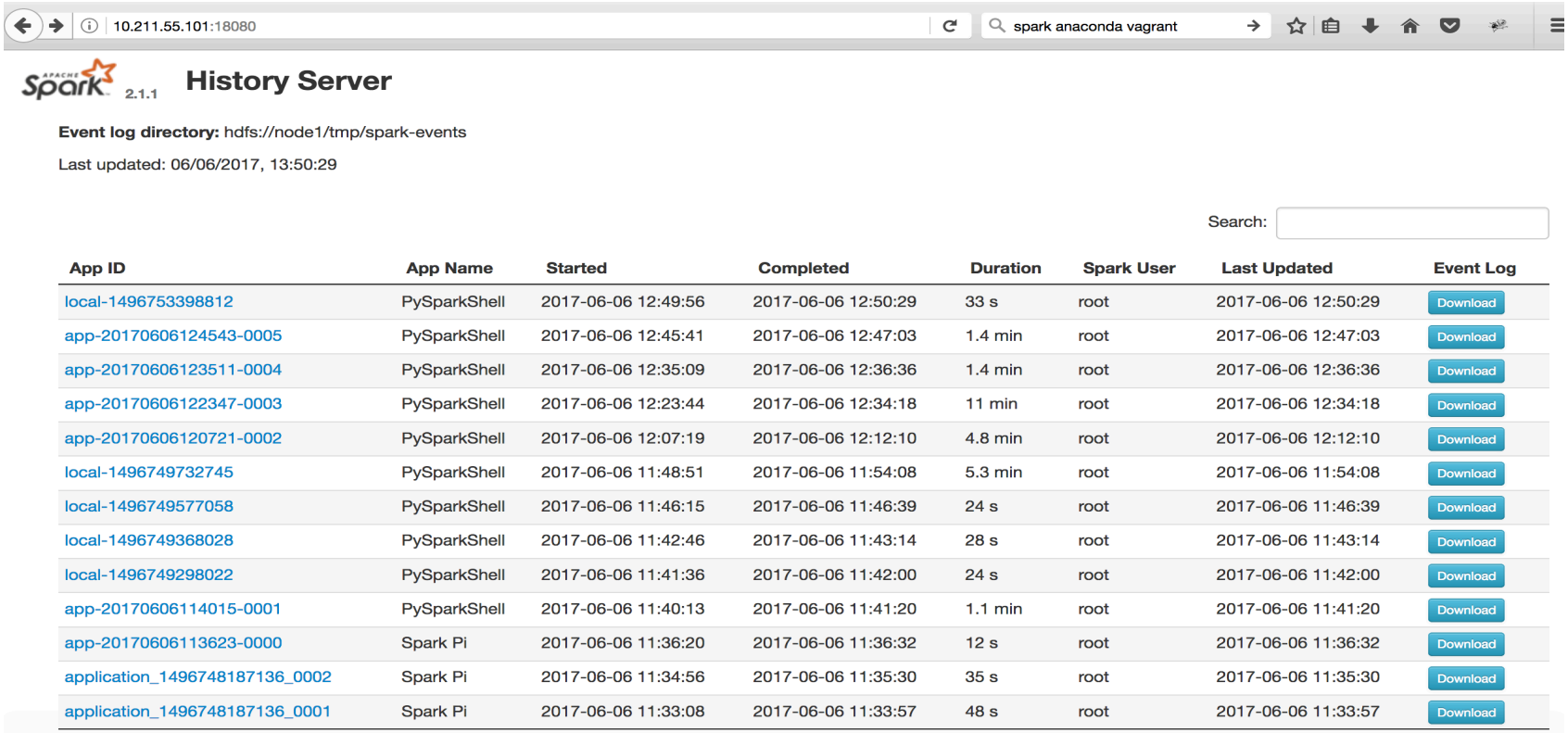
HDFS NameNode UI



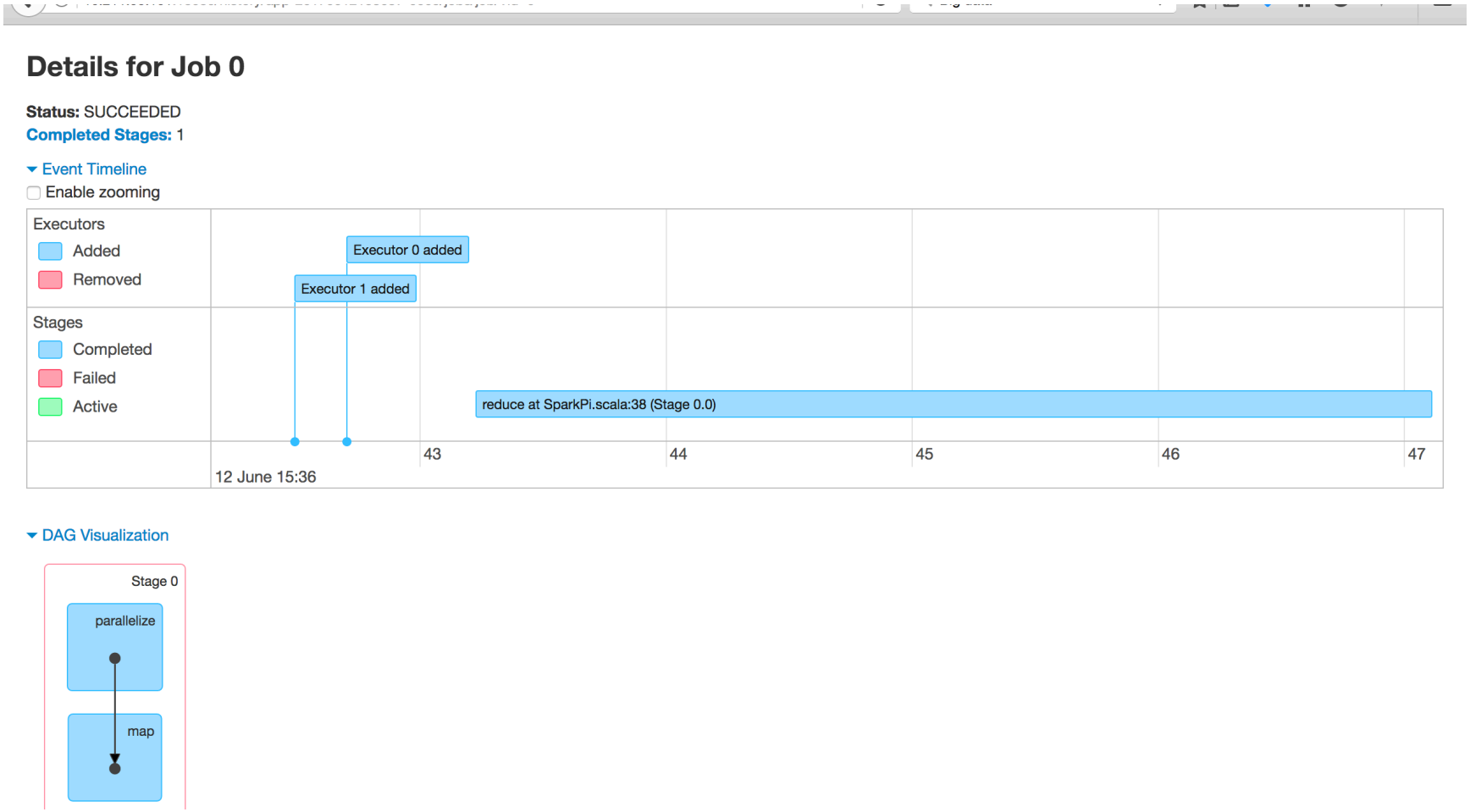
Resource Manager UI



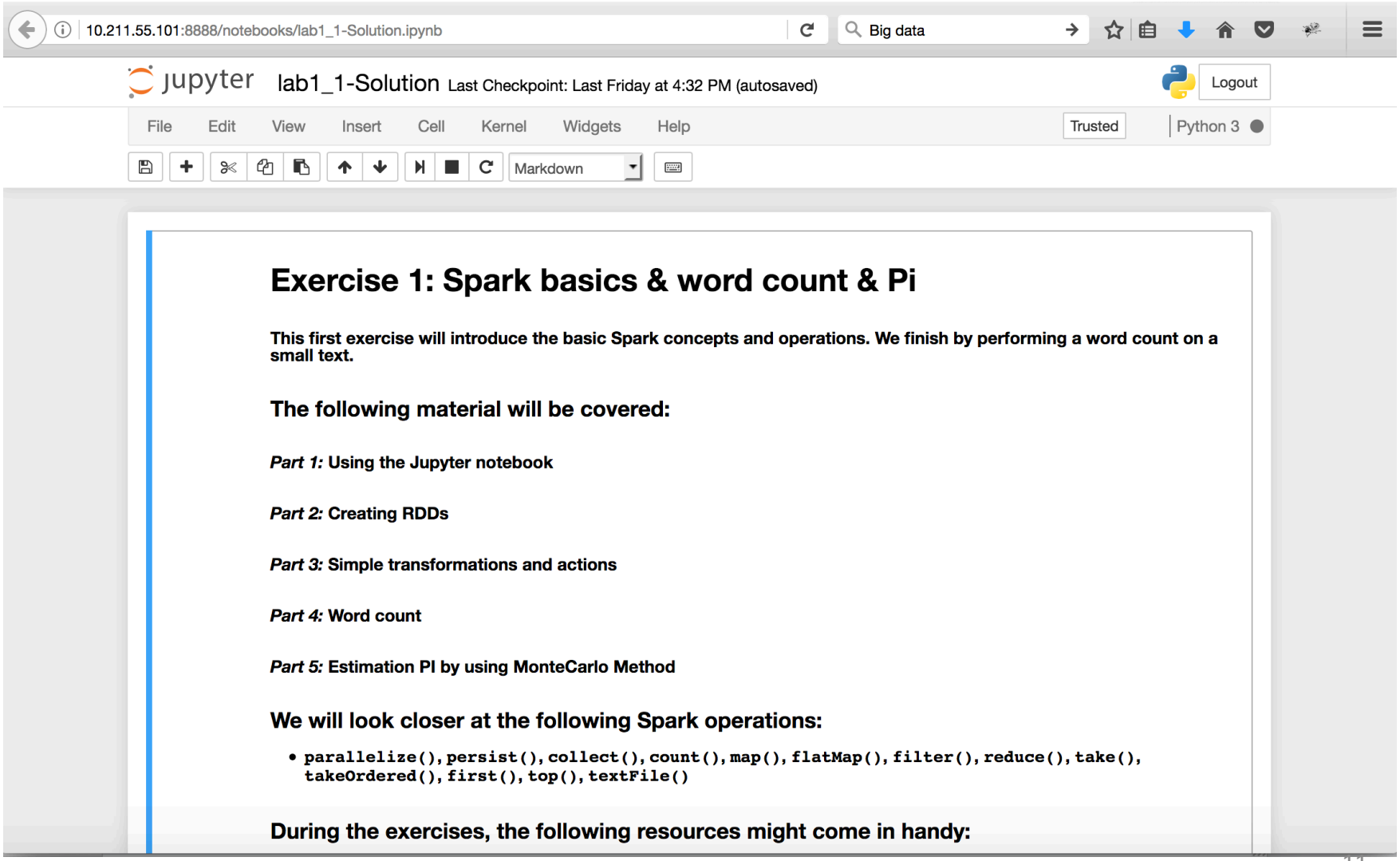
Spark History UI



Spark History UI



Jupyter Notebook



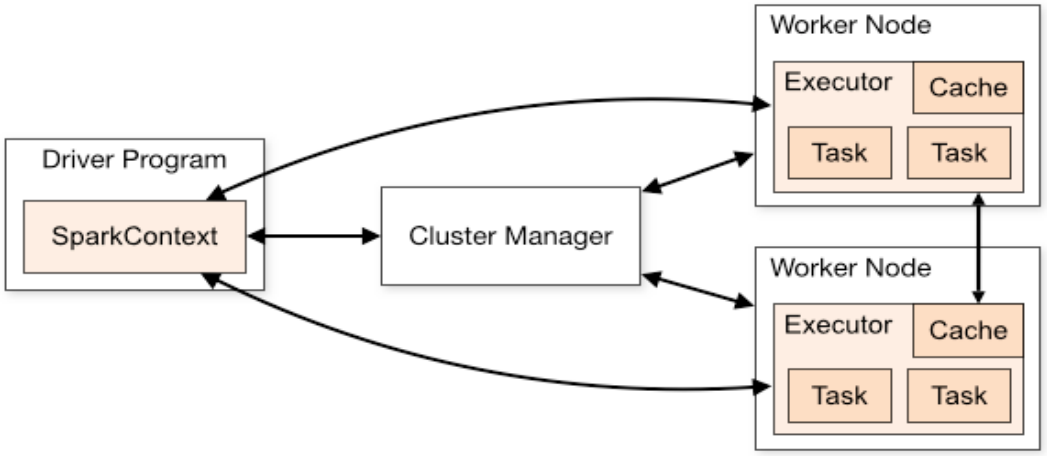
Running Spark Applications

- **Notebooks** are great for:
  - developing and testing quickly experiment with the data
  - demos and collaborating with other people

- **Spark-submit** jobs are more likely to be used in **production**.

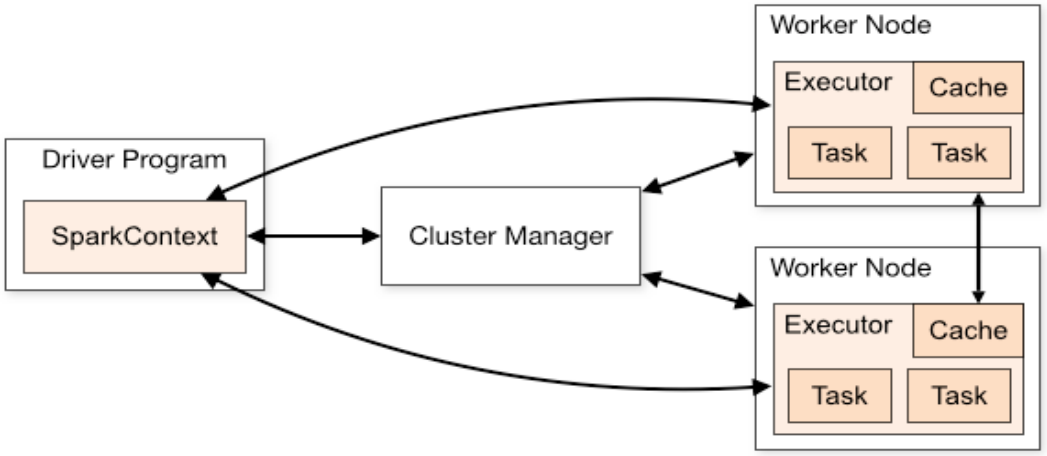
Spark Execution

- Spark applications are run as independent sets of processes, coordinated by a SparkContext in a *driver* program.
- The context will connect to some cluster manager (e.g. YARN, Standalone, Local) which allocates system resources.
- Each worker in the cluster is managed by an *executor*, which is in turn managed by the SparkContext.
- The executor manages computation as well as storage and caching on each machine.

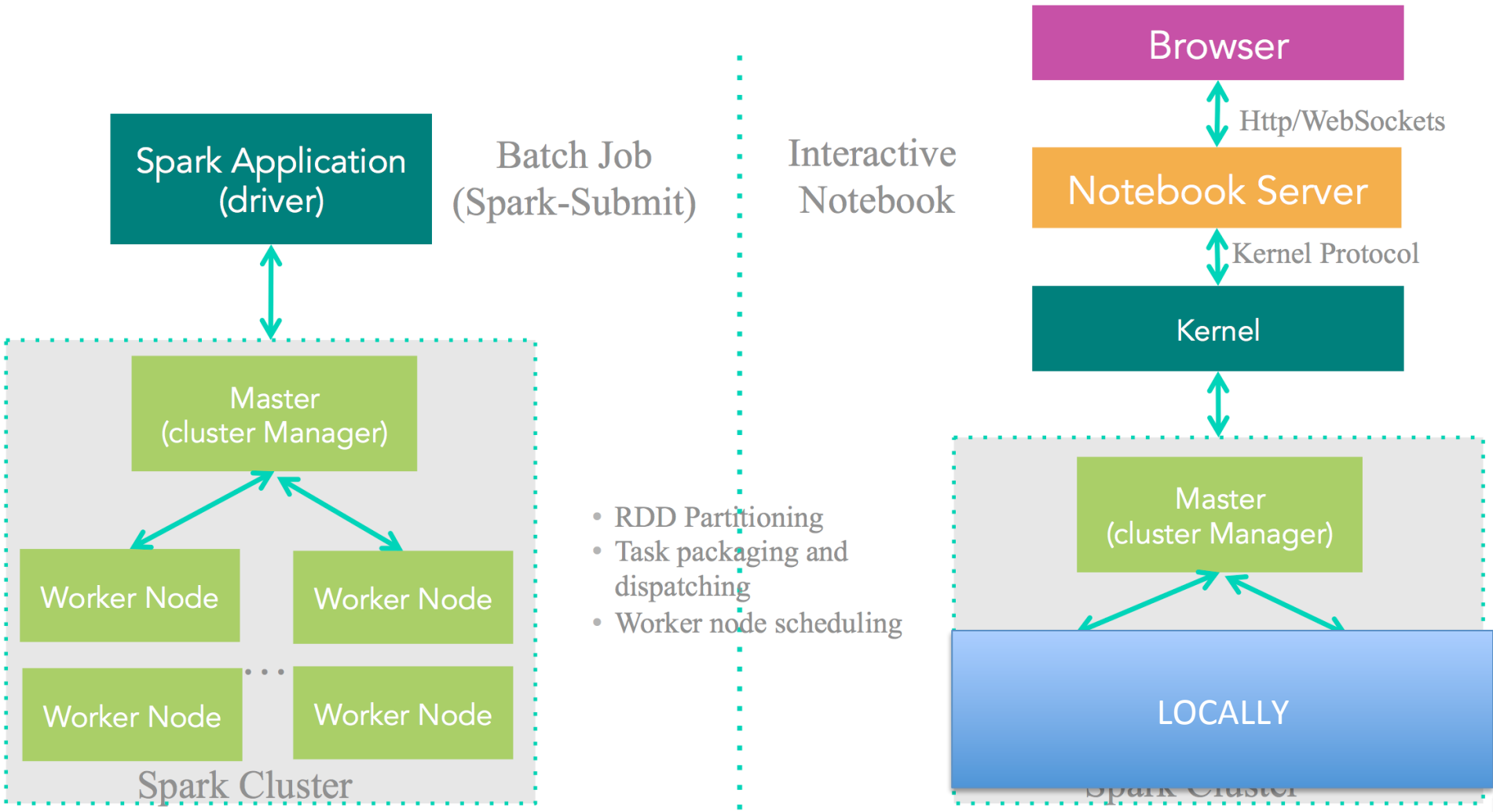


Spark Execution

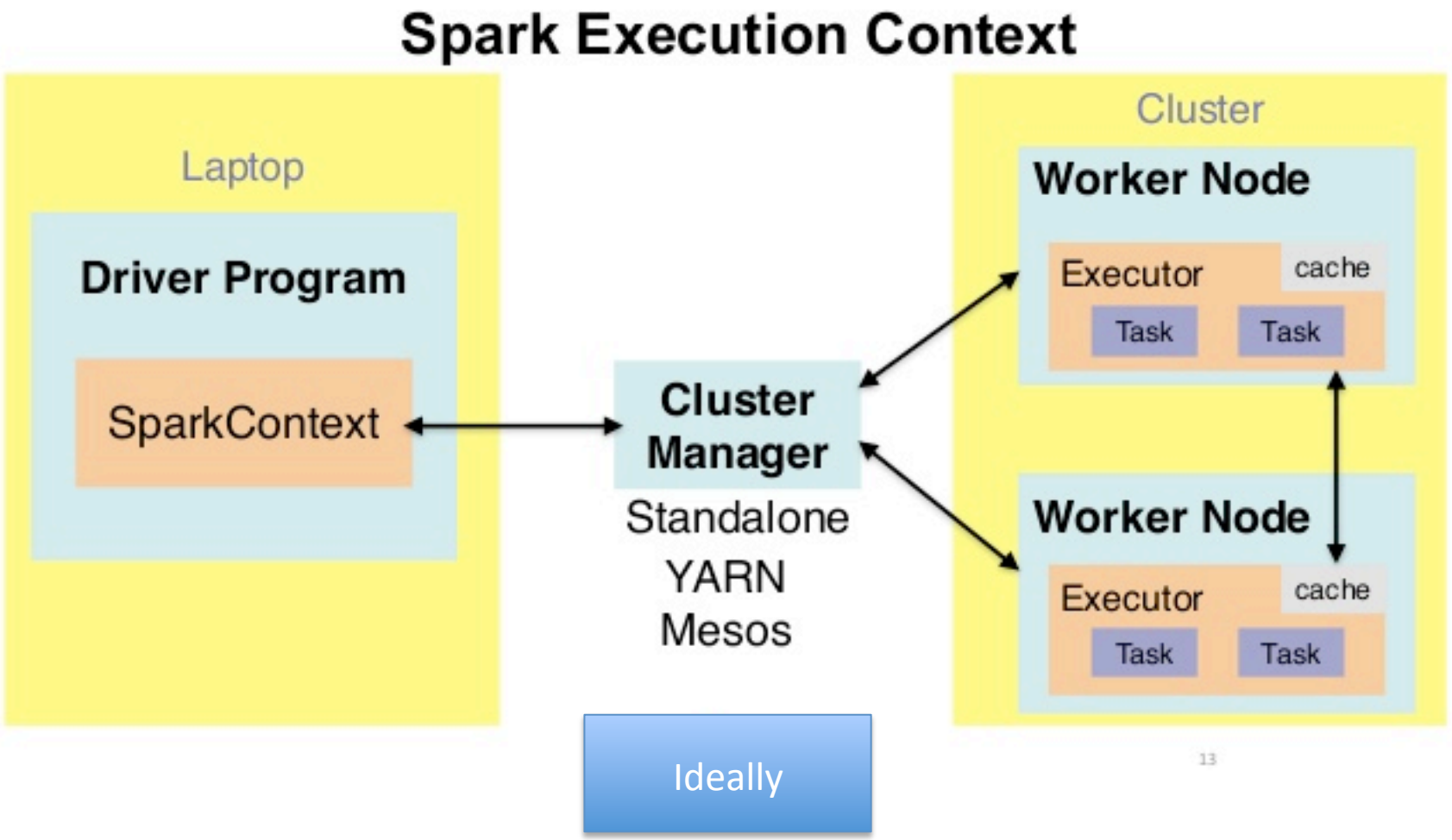
- What is important to note is that application code is sent from the driver to the executors, and the executors specify the context and the various *tasks* to be run.
- The executors communicate back and forth with the driver for data sharing or for interaction.
- Drivers are key participants in Spark jobs, and therefore, they should be on the same network as the cluster



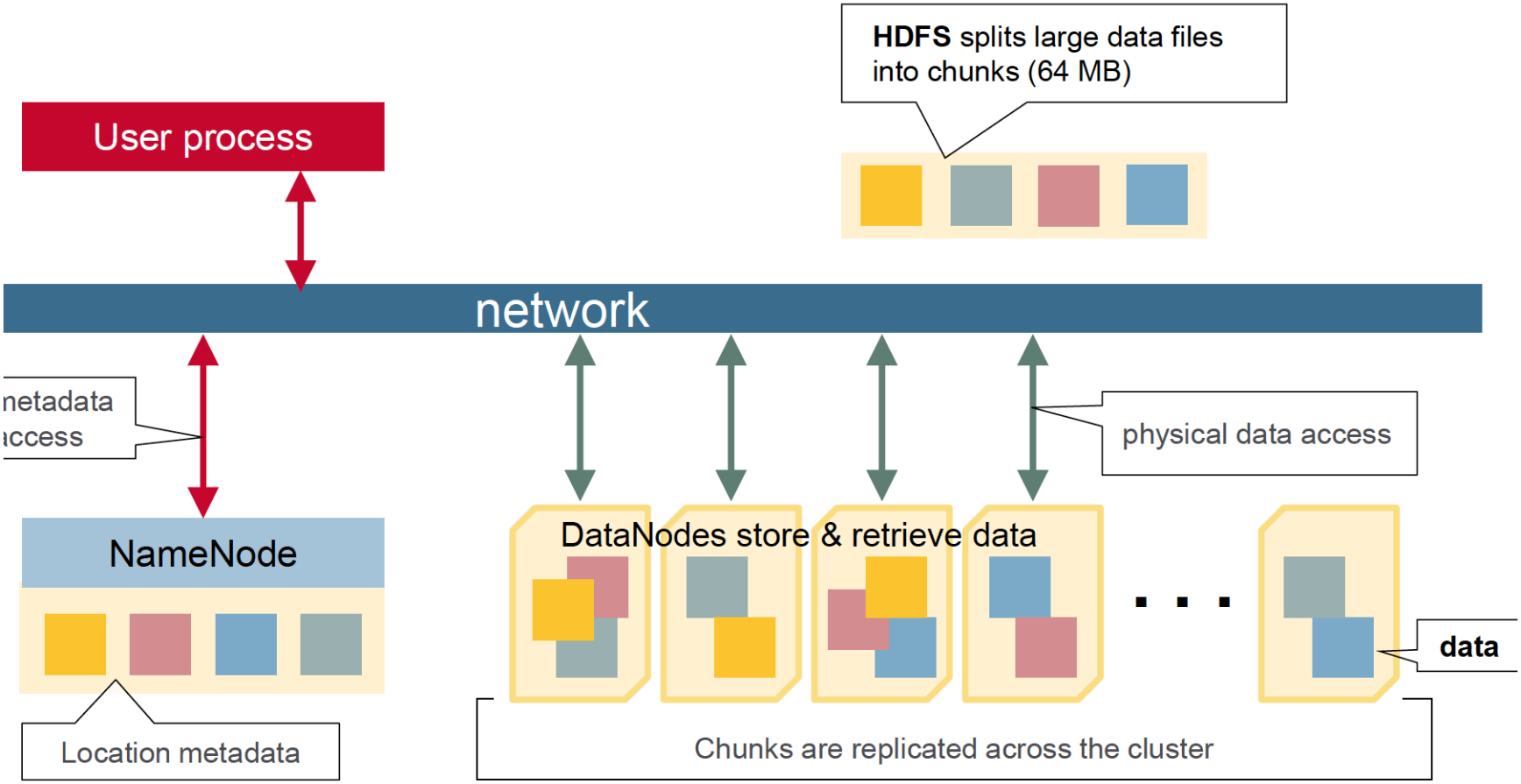
Running Spark Applications



Another possibility



HDFS


HDFS splits large data files into chunks (64 MB)
User process
network
netadata
ccess
physical data access
NameNode
DataNodes store & retrieve data
data
Location metadata
Chunks are replicated across the cluster

HDFS

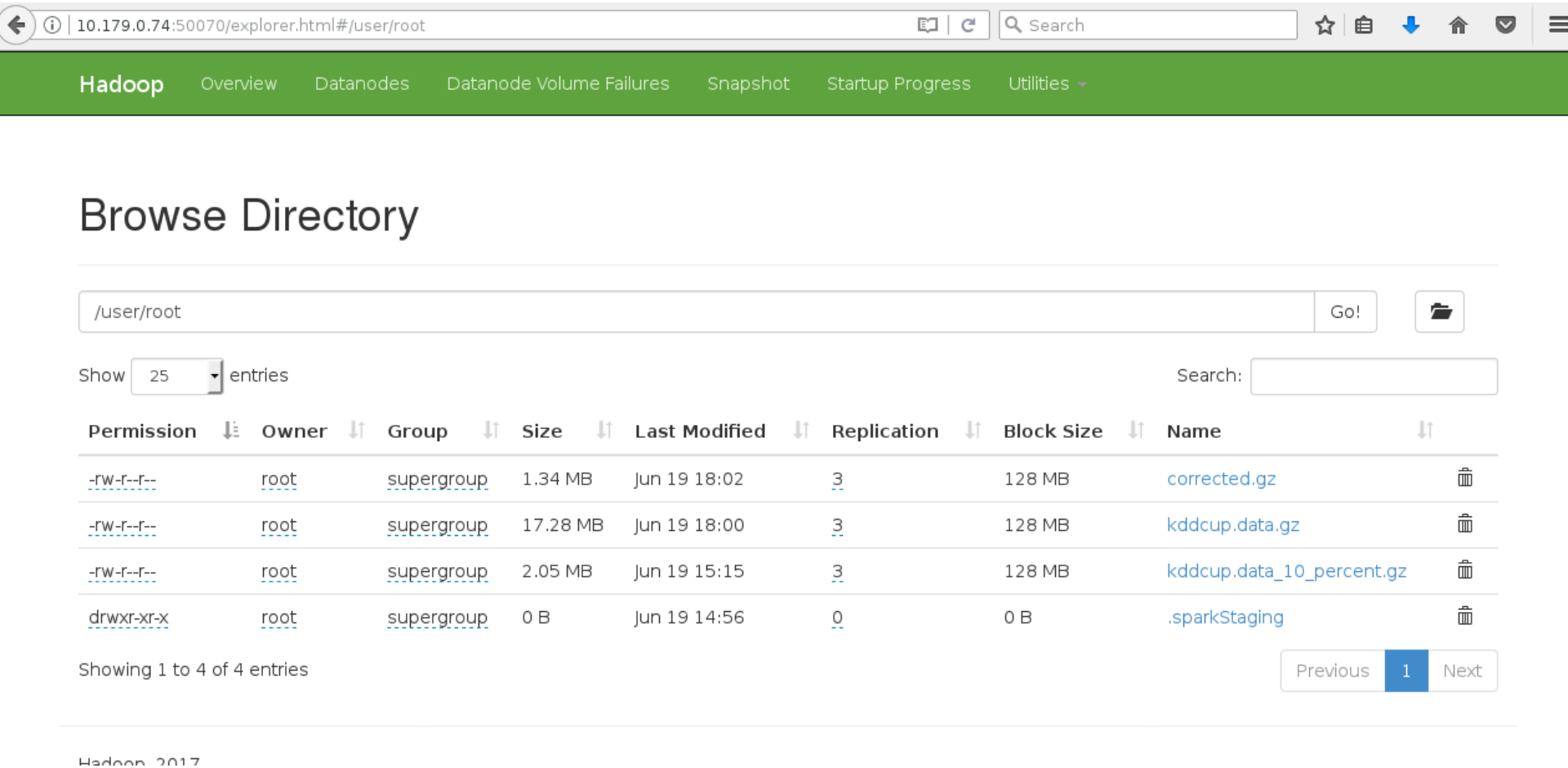
- Hadoop Distributed File System
- The primary command is: **hdfs dfs**
- Has a lot of basic commands
- Copy a local file to HDFS:

  [-put [-f] [-p] [-l] [-d] <localsrc> ... <dst>]

- Assigment:
  - Type the following command in your shell to copy the Readme File into HDFS

```
>> hdfs dfs –put “corrected.gz” “corrected.gz”
```

HDFS NameNode UI



HDFS

Common commands

- **-ls [path]**
  list directories, if no path is specified it assumes your home directory: /user/[username]
- **copyFromLocal [local] [hdfs]**
  copy data from the local filesystem to HDFS
- **-copyToLocal [hdfs] [local]**
  copy data from HDFS to the local filesystem
- **-cat [hdfs]**
  show the contents of files directly from HDFS
- **-mv, -cp, -df, -du**
  move, copy, disk free, disk used
- **-rm, -rmdir, -expunge**
  removal commands

Spark Application

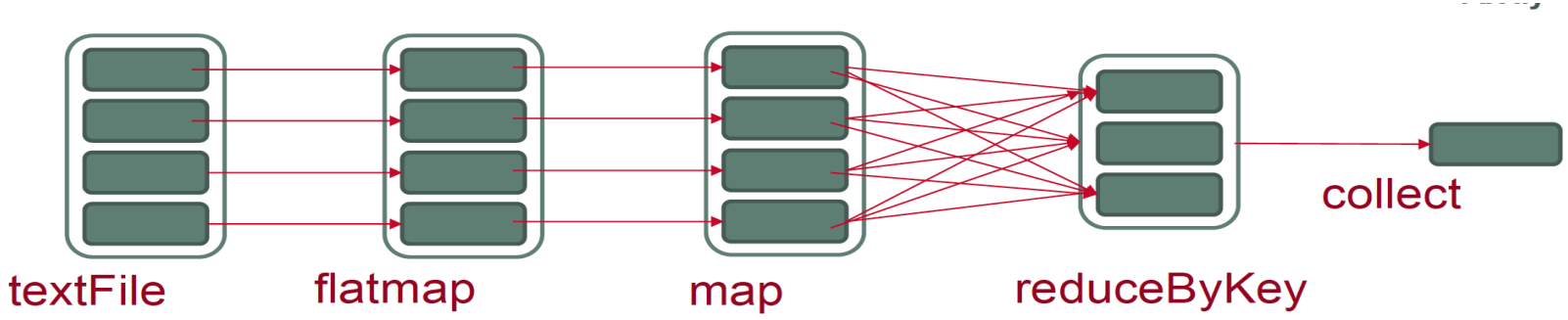
The application that we are going to create is a simple word count:

1. Performs a *textFile* operation to read an input file in HDFS
2. *flatMap* operation to split each line into words
3. *map* operation to form (word, 1) pairs
4. *reduceByKey* operation to sum the counts for each word.

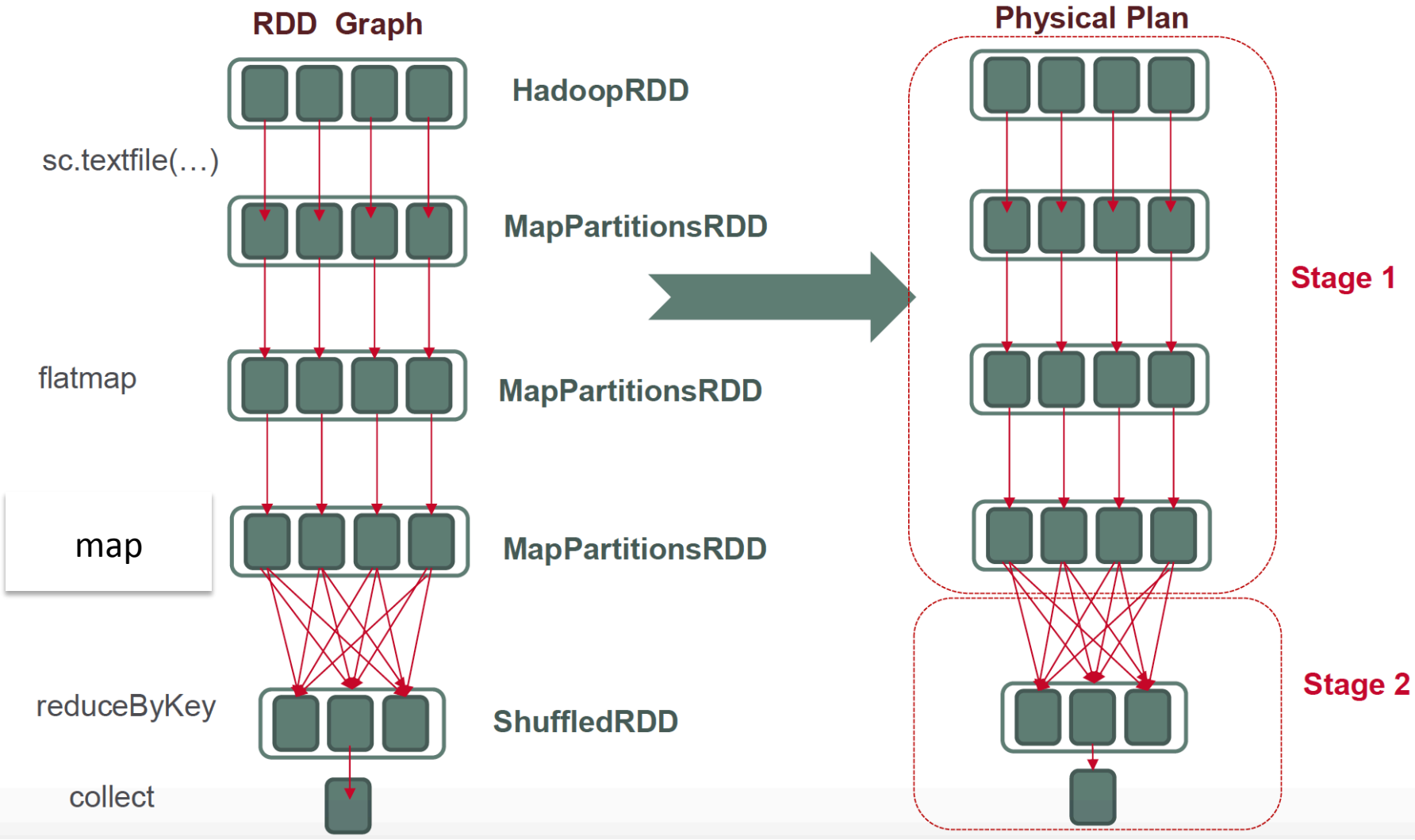
Spark Application

```
import sys
from pyspark import SparkContext, SparkConf


if __name__ == "__main__":
    conf = SparkConf().setAppName("Spark Count")
    sc = SparkContext(conf=conf)
    logFile = "README.md"
    textFile = sc.textFile(logFile)
    wordCounts = textFile.flatMap(lambda line:
    line.split()).map(lambda word: (word, 1)).reduceByKey(lambda a,
    b: a+b)
    numWords=wordCounts.collect()
```

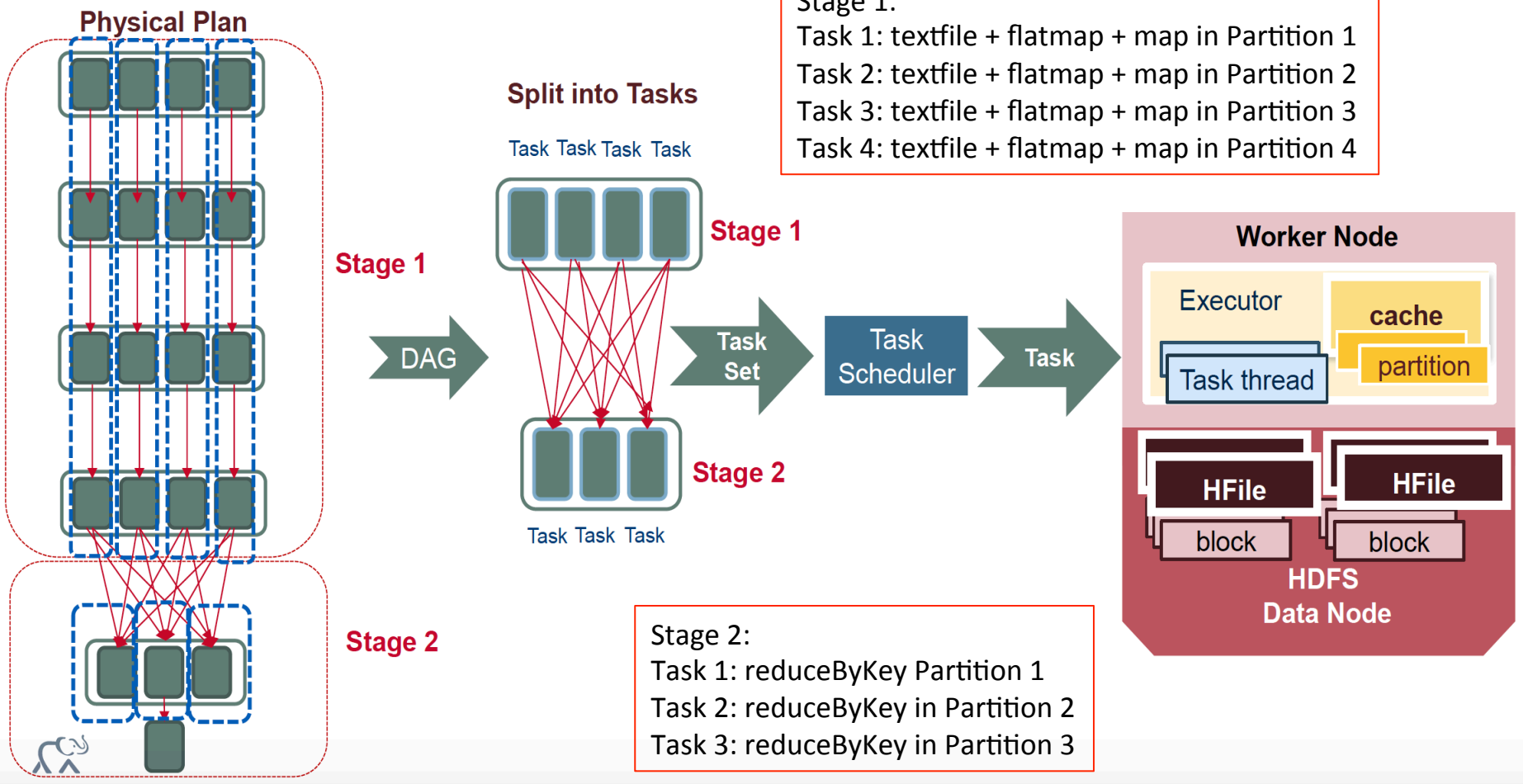


Spark RDD DAG -> Physical Execution plan



Initial RDD distributed among 4 partitions. Final RDD distributed among 3 partitions

Physical Execution plan -> Stages and Tasks

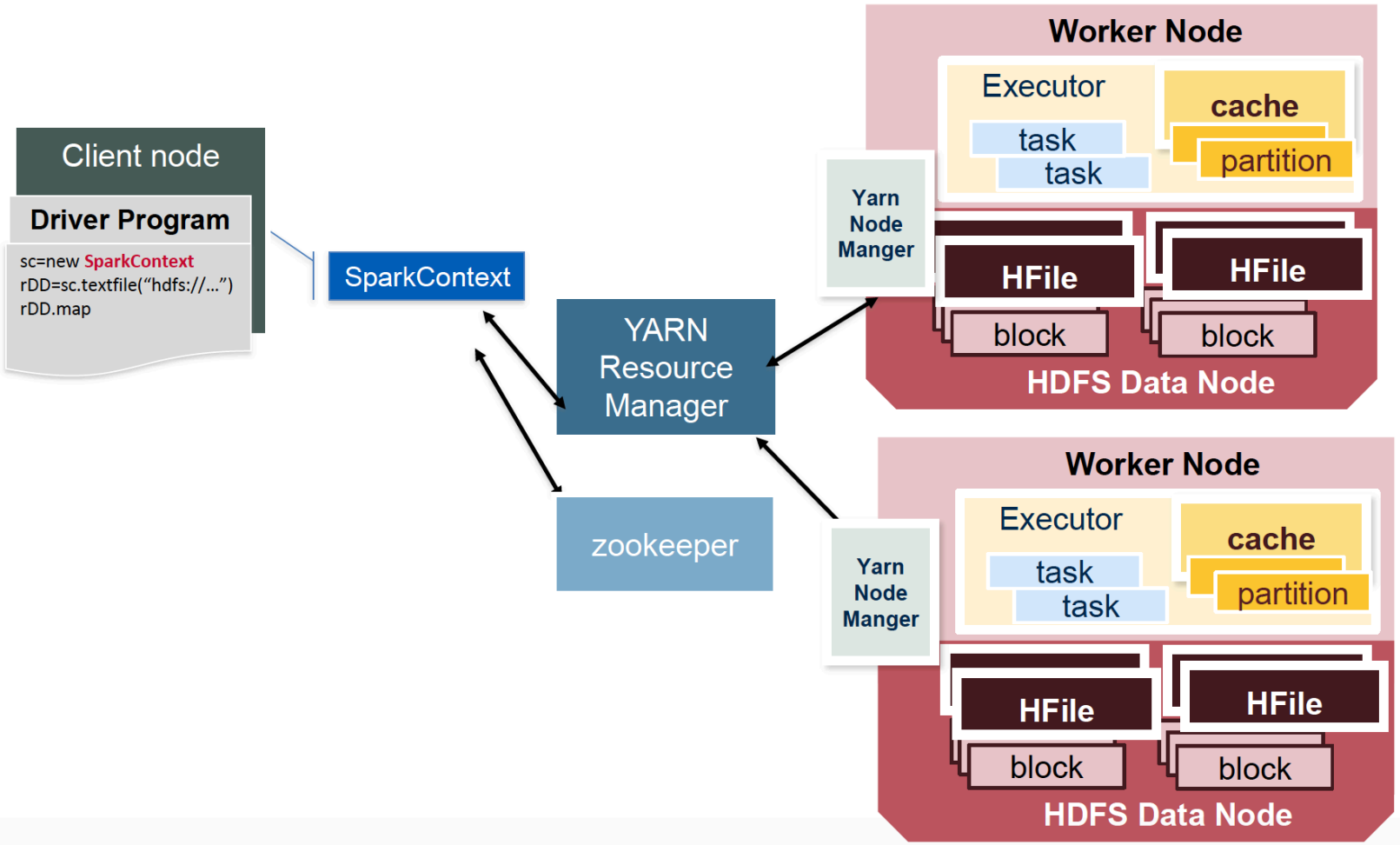


Operations that can run on the same partition are executed in stages

Spark Components

- Job : set of tasks executed as a result of an action
- Stage: set of tasks in a job that can be executed in parallel
- Task: individual unit of work sent to one executor over a sequences of partitions
- Dag: Logical Graph of RDD operations
- RDD: Parallel dataset with partitions

Spark Application on Hadoop cluster with YARN



Submit job via spark-submit

spark-submit Syntax

```
spark-submit --option value \
  application jar | python file [application arguments]
```

Submit job via spark-submit

```
$SPARK_HOME/bin/spark-submit \
--class <main-class> \
--master <master-url> \
--deploy-mode <deploy-mode>  \
--conf \
….
<application-jar> [arguments] |
<python file >[arguments]
```

Some spark-submit options

- master – Determines how to run the job:
  - 'yarn-cluster'
  - **spark://node1:7077**
  - Local
- driver-memory
  - amount memory available for the driver process.
- executor-memory
  - amount of memory allocated to the executor process
- executor-cores
  - total number of cores allocated to the executor process

Note: Complete list at https://spark.apache.org/docs/latest/submitting-applications.html

Submit our application via spark-submit

- Yarn-cluster: $SPARK_HOME/bin/spark-submit --**master yarn-cluster** wordcount.py
- Standalone: : $SPARK_HOME/bin/spark-submit --**master spark://node1:7077** wordcount.py
- Local: $SPARK_HOME/bin/spark-submit --**master local** wordcount.py

- And check the Spark History UI to see the results:
  - http://10.211.55.101:18080

Understanding your Apache Spark Application Through Visualization

- Have a look to this website to understand the Spark History UI: https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html