*E6893 Big Data Analytics*

# HOMEWORK 1
# **TUTORIAL**

COLUMBIA ENGINEERING
The Fu Foundation School
of Engineering and Applied Science

# QUESTION 1 - PROBLEM SETUP

Implement iterative K-means in Spark. We've provided you with starter code kmeans.py on Canvas, which takes care of data loading. **Complete the logic inside the for loop, and run the code with different initialization strategies and loss functions.** Feel free to change the code if needed, or paste the code into a Jupyter notebook. Take a screenshot of your code and results in your report

(1) Run clustering on data.txt with c1.txt and c2.txt as initial centroids and use **L1 distance as similarity measurement**. Compute and plot the within-cluster cost for each iteration. You'll need to submit **two graphs here.**

(2) Run clustering on data.txt with c1.txt and c2.txt as initial centroids and use **L2 distance as similarity measurement**. Compute and plot the within-cluster cost for each iteration. You'll need to submit two graphs here.

(3) t-SNE is a dimensionality reduction method particularly suitable for visualization of high-dimensional data. Visualize your clustering assignment result of (2) by reducing the dimension to a 2D space. You'll need to submit two graphs here.

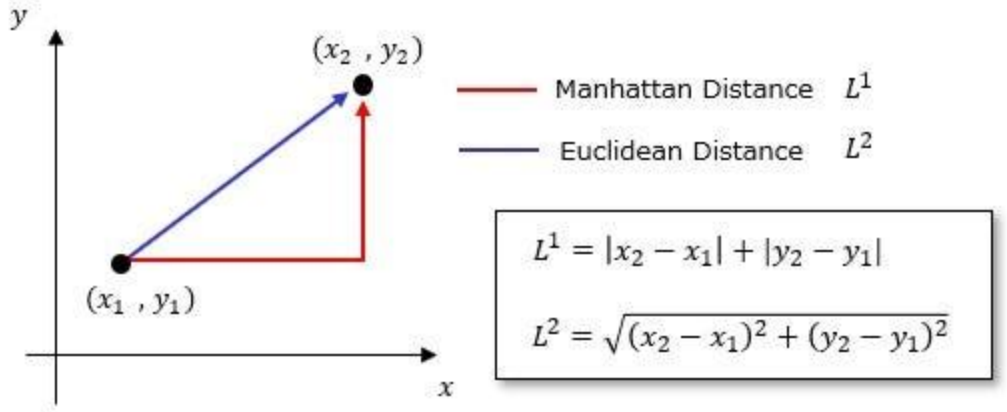
(4) For L2 and L1 distance, are random initialization of K-means using c1.txt better than initialization using c2.txt in terms of cost? Explain your reasoning.

(5) What is the time complexity of the iterative K-means?

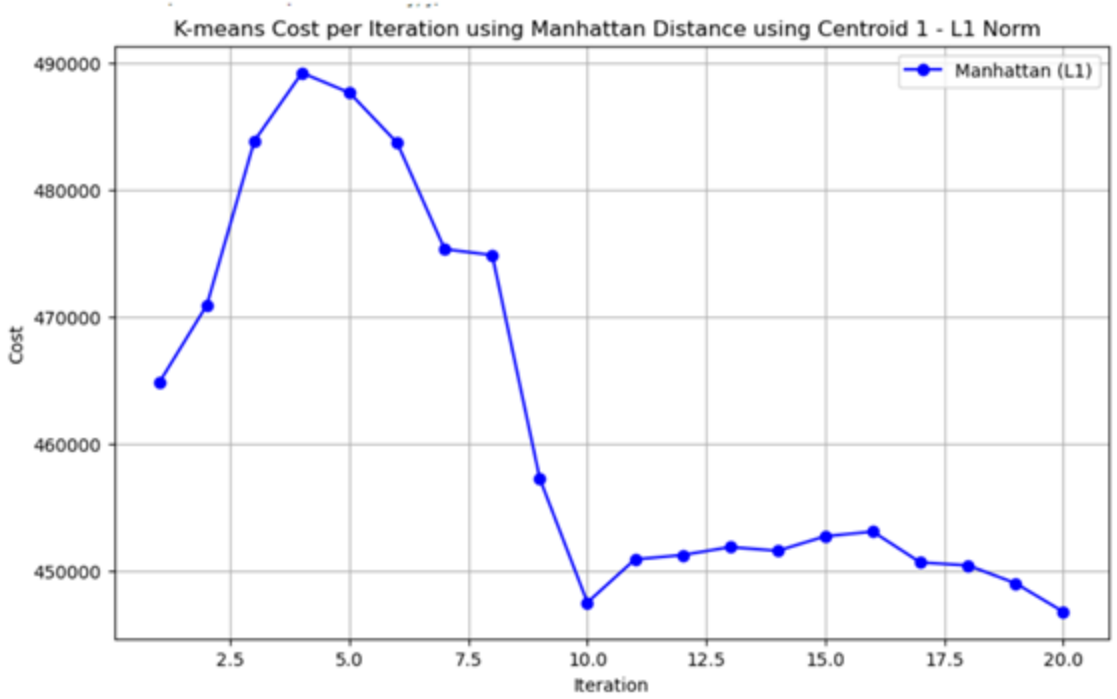
# QUESTION 1 - PART 1

**Restate the task**

- **Input:**
  - `data.txt` → 4601 docs × 58 features
  - `c1.txt` and `c2.txt` → two sets of initial centroids
- **Distance function:** L1 (Manhattan)
- **Iterations:** 20, with k = 10
- **Output:** Plot within-cluster cost vs. iteration (two graphs: c1-init, c2-init)




**Different from L2: adds absolute differences (linear growth) !**

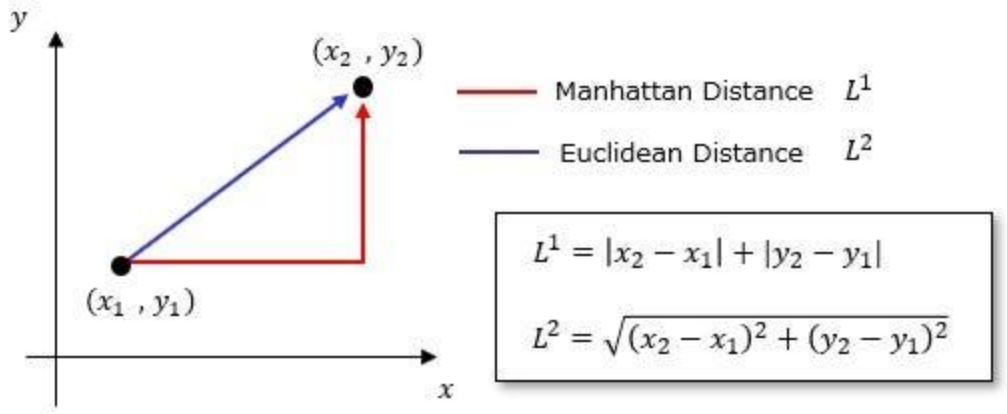



(please don't use this as your answer)

# QUESTION 1 - PART 1

**Restate the task**

- **Input:**
  - `data.txt` → 4601 docs × 58 features
  - `c1.txt` and `c2.txt` → two sets of initial centroids
- **Distance function:** L1 (Manhattan)
- **Iterations:** 20, with k = 10
- **Output:** Plot within-cluster cost vs. iteration (two graphs: c1-init, c2-init)




For **each iteration (1 to N):**

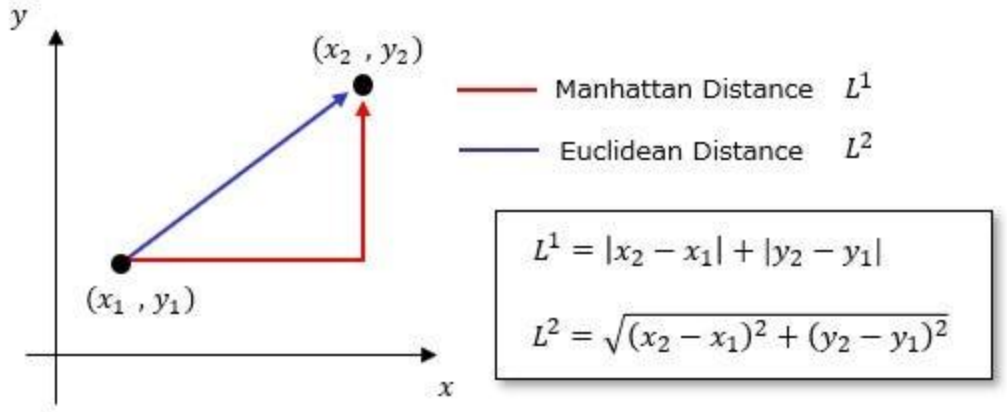
1. Assign each document → nearest centroid using **L1**
   (Spark: `map` each point → cluster ID)
2. Compute **cost**: sum of distances of each point to its assigned centroid.
   (Spark: `map` distances → `reduce` to total)
3. Recompute centroids (mean of assigned points, unless you implement true median for L1).

# QUESTION 1 - PART 2

**Restate the task / Similar to L1 Implementation**

- **Input:**
  - `data.txt` → 4601 docs × 58 features
  - `c1.txt` and `c2.txt` → two sets of initial centroids

- **Distance function:** L2 (Euclidean)
- **Iterations:** 20, with k = 10
- **Output:** Plot within-cluster cost vs. iteration (two graphs: c1-init, c2-init)

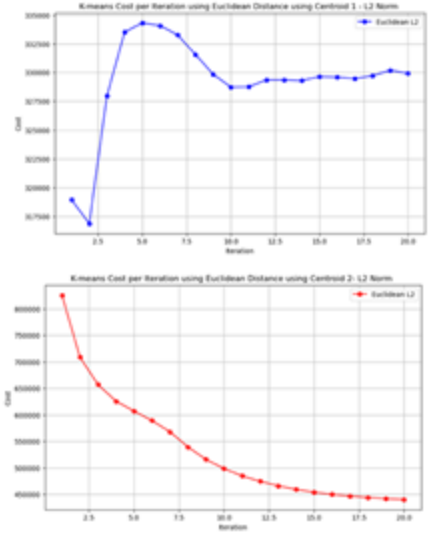



For **each iteration (1 to N):**

1. Assign each document → nearest centroid using **L2**
2. (Spark: `map` each point → cluster ID)

3. Compute **cost**: sum of distances of each point to its assigned centroid.
   (Spark: `map` distances → `reduce` to total)

4. Recompute centroids (mean of assigned points, unless you implement true median for L1).

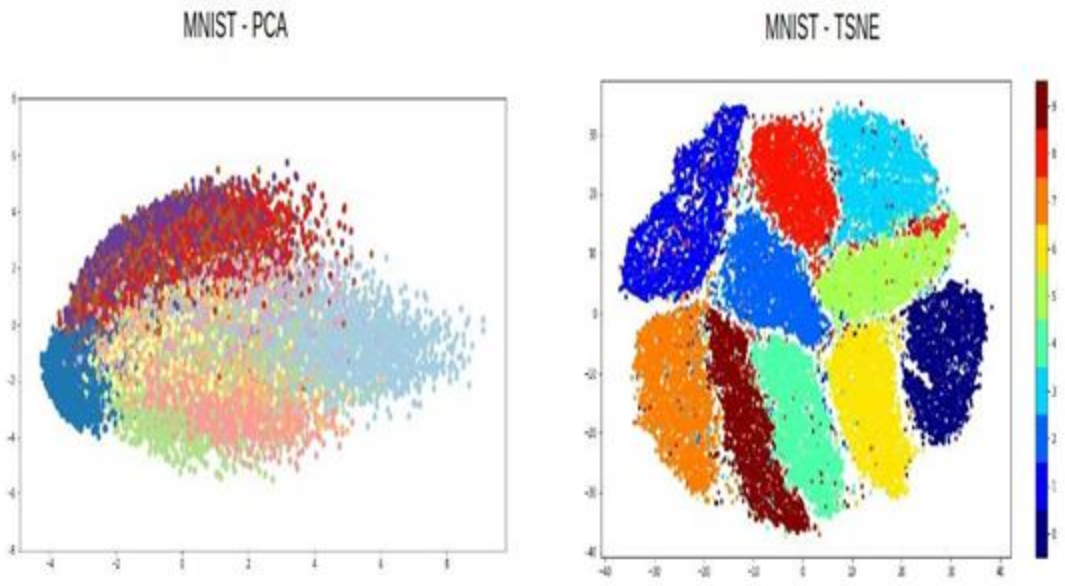


# QUESTION 1 - PART 3

t-SNE : t-distributed Stochastic Neighbor Embedding

A **dimensionality reduction** algorithm (like PCA), but optimized for **visualization**.

Take **high-dimensional data** (58-D documents here) and show it in **2D** (or 3D) while **preserving neighborhood structure**.




| Feature | PCA | t-SNE |
|---|---|---|
| Type | Linear | Non-linear |
| Goal | Capture max variance | Preserve local neighborhoods |
| Good for | Dimensionality reduction before ML | Visualization (2D/3D) |
| Global vs Local | Preserves global structure | Preserves local structure |
| Speed | Fast | Slower (iterative optimization) |
| Output | Any # of dims | Usually 2D or 3D only |

# QUESTION 2 - BINARY CLASSIFICATION ON SPARK

**Frame the problem**

- Dataset: **Adult (Census Income)** ~48k rows.
  Goal: predict **income ≥ 50K? → binary classification**.
- Deliverables: **load → preprocess → train 9 models → evaluate & rank accuracy**.

**Data loading**

- Read CSV as **DataFrame** with `inferSchema` and header handling.
- **Rename columns** to:

```
["age","workclass","fnlwgt","education","education_num","marital_status","occupatio
n","relationship","race","sex","capital_gain","capital_loss","hours_per_week","nati
ve_country","income"]
```

- Check **schema** & **nulls/'?'**.

- "Which columns look categorical vs numerical?"
- "What should we do with '?' entries?"

# QUESTION 2 - BINARY CLASSIFICATION ON SPARK

**Preprocessing**

**Goal:** turn mixed schema → single **features** vector + **label**.

1. **Clean string columns** (trim/lowercase; replace `?` with “Unknown”).
2. **StringIndexer** on each categorical col (fit on **train only**!).
3. **OneHotEncoder** on indexed cats (dropLast=true).
   **Assemble** numeric + OHE categorical into `features`.
4. **Label**: index `income` → `{0, 1}` (e.g., `<=50K`→0, `>50K`→1).
5. **Split**: 70/30, `seed=100`.

**Some Questions?**

- “Why index *before* one-hot?”
- “Why assemble after encoding?”
- “Why split before fitting transformers?” (data leakage)

# QUESTION 2 - BINARY CLASSIFICATION ON SPARK

**Modeling (9 estimators)**

**The 9 models (Spark MLlib)**

1. **Logistic Regression (LR)** – linear decision boundary; probs via sigmoid.
   Hyperparams to mention: `regParam`, `elasticNetParam`, `maxIter`.
2. **Random Forest (RF)** – ensemble of trees; handles nonlinearity; robust.
   `numTrees`, `maxDepth`, `featureSubsetStrategy`.
   **Naive Bayes (NB)** – assumes feature independence; best with counts; try `multinomial`.
3. **Decision Tree (DT)** – interpretable; prone to overfit; `maxDepth`, `minInstancesPerNode`.
4. **Gradient-Boosted Trees (GBT)** – strong accuracy; `maxIter`, `maxDepth`. (Binary only.)
5. **Multilayer Perceptron (MLP)** – simple feedforward NN; specify **layers** like `[input, h1, …, 2]`.
6. **Linear SVM (LSVC)** – margin-based linear classifier; fast; no probabilities by default.
   **One-vs-Rest (OvR)** – meta-wrapper for multi-class; here it will just wrap a base linear model

**Reuse the same preprocessed `features` for all models.**

Wrap each model in a small **Pipeline**: `[indexers → encoder → assembler → estimator]`

# QUESTION 2 - BINARY CLASSIFICATION ON SPARK

**Modeling (9 estimators)**

**The 9 models (Spark MLlib)**

1. **Logistic Regression (LR)** – linear decision boundary; probs via sigmoid.
   Hyperparams to mention: `regParam`, `elasticNetParam`, `maxIter`.
2. **Random Forest (RF)** – ensemble of trees; handles nonlinearity; robust.
   `numTrees`, `maxDepth`, `featureSubsetStrategy`.
   **Naive Bayes (NB)** – assumes feature independence; best with counts; try `multinomial`.
3. **Decision Tree (DT)** – interpretable; prone to overfit; `maxDepth`, `minInstancesPerNode`.
4. **Gradient-Boosted Trees (GBT)** – strong accuracy; `maxIter`, `maxDepth`. (Binary only.)
5. **Multilayer Perceptron (MLP)** – simple feedforward NN; specify **layers** like `[input, h1, …, 2]`.
6. **Linear SVM (LSVC)** – margin-based linear classifier; fast; no probabilities by default.
   **One-vs-Rest (OvR)** – meta-wrapper for multi-class; here it will just wrap a base linear model

**Reuse the same preprocessed `features` for all models.**

Wrap each model in a small **Pipeline**: `[indexers → encoder → assembler → estimator]`

# QUESTION 3 - HADOOP

**Big Picture – What is Hadoop?**

**Slide / Whiteboard Bullets**

- **Hadoop Common**
  - Shared utilities, config files, JARs, startup scripts
  - The “foundation layer” supporting all modules
- **HDFS (Hadoop Distributed File System)**
  - Stores data across multiple machines
  - Fault-tolerant (replication of blocks)
  - High-throughput reads/writes
- **YARN (Yet Another Resource Negotiator)**
  - Cluster resource manager
  - Handles scheduling and allocation of CPU, memory
  - Lets multiple apps share the same cluster
- **MapReduce**
  - Parallel data processing framework
  - Breaks jobs into map + reduce tasks
  - Runs on top of YARN

# QUESTION 3 - HADOOP

Hadoop is not just one thing. It's a framework made up of **four core modules**

Think of **Hadoop Common** as the plumbing: the basic utilities and libraries all the other parts need

**HDFS** is the storage layer. Instead of putting one giant file on a single server, it splits the file into blocks and spreads them across machines. If one machine fails, copies exist elsewhere.

**YARN** is the traffic cop of the cluster. It decides which job gets how many CPUs, how much memory, and where tasks will run.

Finally, **MapReduce** is the compute engine. It processes data in parallel by dividing the job into small tasks and combining results

**So, Which part handles storage? Which handles computation? Where does scheduling fit?**

- **Storage = HDFS**
- **Computation = MapReduce**
- **Scheduling/resources = YARN**

# QUESTION 3 - HADOOP - PART 1

Part (1): Download & Setup Hadoop

**Provide screenshots of each step in the "Verify Hadoop installation" section of the tutorial.**

**screenshots** of successful web UIs

Part (2): HDFS Metrics Monitoring

**Where to find HDFS metrics ????**

## NameNode in Hadoop

The **NameNode** is the **master server** of HDFS (Hadoop Distributed File System).

It manages the **metadata** of the file system:

- Directory structure (like a file system tree)
- File names and locations
- Which blocks make up each file
- Which DataNodes store those blocks

**It does not store the actual file data only information about where the data lives !**

# QUESTION 3 - HADOOP - PART 1

## NameNode in Hadoop

The **NameNode** is the **master server** of HDFS (Hadoop Distributed File System).

It manages the **metadata** of the file system:

- Directory structure (like a file system tree)
- File names and locations
- Which blocks make up each file
- Which DataNodes store those blocks

**It does not store the actual file data only information about where the data lives !**

**Role in HDFS**

You can think HDFS like a **library**:

- The **bookshelves** are the **DataNodes** (they store the actual books = data blocks)
- The **librarian's catalog** is the **NameNode** (it knows which shelf each book is on)

# QUESTION 3 - HADOOP - PART 1

Step 1. Access HDFS Metrics via HTTP API

Make sure Hadoop is running in **pseudo-distributed mode**
**verify with jps**

Open the **NameNode web UI** in your browser
**http://localhost:9870/**

Access the **metrics API (JMX endpoint)**:
**http://localhost:9870/jmx**

This returns a **JSON file** with all HDFS metrics
You can view it directly in the browser or save it with `curl`

```
curl http://localhost:9870/jmx > hdfs_metrics.json
```

# QUESTION 3 - HADOOP - PART 1

Step 2. Choose 5 Important Metrics

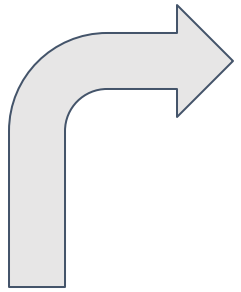
1. **CapacityTotal**
   - *Definition:* Total storage capacity of the HDFS cluster.
   - *Why important:* Tells you the maximum size of data your cluster can hold.

2. **CapacityUsed**
   - *Definition:* Amount of space currently in use.
   - *Why important:* Helps track cluster utilization and avoid running out of storage.

3. **NumLiveDataNodes**
   - *Definition:* Number of DataNodes that are active and responding.
   - *Why important:* Ensures the cluster has enough nodes available for storage and reliability.

4. **MissingBlocks**
   - *Definition:* Number of data blocks that don't have the required number of replicas.
   - *Why important:* Directly impacts **data reliability** , missing blocks mean potential data loss.

5. **CorruptBlocks**
   - *Definition:* Number of corrupted blocks that cannot be read.
     *Why important:* Indicates disk errors or hardware failures; critical for data integrity.



Please try to cover different metrics :)


# EECS E6893 Big Data Analytics
# Spark Dataframe, Spark SQL, Hadoop metrics

# Agenda

- Spark Dataframe
- Spark SQL
- Hadoop metrics

# Spark Dataframe

- An ***abstraction***, an immutable distributed collection of data like RDD
- Data is organized into named columns, like a table in DB
- Create from RDD, Hive table, or other data sources
- Easy conversion with Pandas Dataframe

# Spark Dataframe: read from csv file

```
# read data from csv into Dataframe
df = spark.read.format("csv").option("header", 'true').load("gs://big_data_ta/data/citibike_stations.csv")
```

```
type(df)
```

```
pyspark.sql.dataframe.DataFrame
```

```
df.show(1)
```

```
+----------+------------------+---------+----------+-----------+--------+-------------+--------+-------------
----------+------------------+-----------------+------------------+----------------+-----------+---------+----
--------+-------------------------+------------------+
|station_id|              name|short_name|  latitude|  longitude|region_id|rental_methods|capacity|eightd_has_key
_dispenser|num_bikes_available|num_bikes_disabled|num_docks_available|num_docks_disabled|is_installed|is_renting|is_r
eturning|eightd_has_available_keys|     last_reported|
+----------+------------------+---------+----------+-----------+--------+-------------+--------+-------------
----------+------------------+-----------------+------------------+----------------+-----------+---------+----
--------+-------------------------+------------------+
|       382|University Pl & E...|  5905.11|40.73492695|-73.99200509|       71|KEY,CREDITCARD|       0|
false|                 0|                0|                 0|               0|      false|    false|       fa
lse|                 false|1970-01-01 00:00:00|
+----------+------------------+---------+----------+-----------+--------+-------------+--------+-------------
----------+------------------+-----------------+------------------+----------------+-----------+---------+----
--------+-------------------------+------------------+
only showing top 1 row
```

# Spark Dataframe: common operations

```
df.printSchema()
```

```
root
 |-- station_id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- short_name: string (nullable = true)
 |-- latitude: string (nullable = true)
 |-- longitude: string (nullable = true)
 |-- region_id: string (nullable = true)
 |-- rental_methods: string (nullable = true)
 |-- capacity: string (nullable = true)
 |-- eightd_has_key_dispenser: string (nullable = true)
 |-- num_bikes_available: string (nullable = true)
 |-- num_bikes_disabled: string (nullable = true)
 |-- num_docks_available: string (nullable = true)
 |-- num_docks_disabled: string (nullable = true)
 |-- is_installed: string (nullable = true)
 |-- is_renting: string (nullable = true)
 |-- is_returning: string (nullable = true)
 |-- eightd_has_available_keys: string (nullable = true)
 |-- last_reported: string (nullable = true)
```

```
df.count()
```

```
843
```

# Spark Dataframe: common operations

```
df.columns
```

```
['station_id',
 'name',
 'short_name',
 'latitude',
 'longitude',
 'region_id',
 'rental_methods',
 'capacity',
 'eightd_has_key_dispenser',
 'num_bikes_available',
 'num_bikes_disabled',
 'num_docks_available',
 'num_docks_disabled',
 'is_installed',
 'is_renting',
 'is_returning',
 'eightd_has_available_keys',
 'last_reported']
```

# Spark Dataframe: common operations

```
df.describe().show()
```

```
+-------+-----------------+---------------+-----------------+-----------------+-------------------+-------------------+------------
------+-------------+-----------------+----------------------+-----------------+-------------------+-----------
-------+------------------+-----------+---------+-----------+----------------------+------------------+
|summary|       station_id|           name|       short_name|         latitude|          longitude|          reg
ion_id|rental_methods|         capacity|eightd_has_key_dispenser|num_bikes_available|num_bikes_disabled|num_docks_av
ailable| num_docks_disabled|is_installed|is_renting|is_returning|eightd_has_available_keys|     last_reported|
+-------+-----------------+---------------+-----------------+-----------------+-------------------+-------------------+------------
------+-------------+-----------------+----------------------+-----------------+-------------------+-----------
-------+------------------+-----------+---------+-----------+----------------------+------------------+
|  count|              843|            843|              843|              843|                843|                843|
843|          843|              843|                   843|              843|                843|
843|               843|        843|      843|        843|                   843|               843|
|   mean| 2434.425860023725|           null| 5806.786515151487| 40.73212559944772|   -73.9749901186049|  70.93950177
935943|          null|31.419928825622776|                  null| 14.565836298932384|0.5693950177935944|  16.1897983
3926453|0.05219454329774614|       null|     null|       null|                  null|              null|
| stddev|1421.1204113008778|           null|1175.6743390458948|0.038745169614834 1|0.031207687758326202|0.23854913482
063428|          null|12.052012437572532|                  null| 11.188256063195926|0.8613434732614029| 13.15807566
4204775| 0.6499620307701626|       null|     null|       null|                  null|              null|
|    min|              116|1 Ave & E 110 St|           3460.01|40.655399774478312|        -73.9077436|
70|KEY,CREDITCARD|                  0|                false|                0|                  0|
0|                0|       false|    false|      false|                 false|1970-01-01 00:00:00|
|    max|               83|         York St|            JC106|40.814394437915816|        -74.0836394|
71|KEY,CREDITCARD|                 79|                 true|                9|                  6|
9|                9|        true|     true|       true|                  true|2019-09-02 00:00:00|
+-------+-----------------+---------------+-----------------+-----------------+-------------------+-------------------+------------
------+-------------+-----------------+----------------------+-----------------+-------------------+-----------
-------+------------------+-----------+---------+-----------+----------------------+------------------+
```

# Spark Dataframe: common operations

```
df.describe('capacity').show()
```

```
+-------+------------------+
|summary|          capacity|
+-------+------------------+
|  count|               843|
|   mean|31.419928825622776|
| stddev|12.052012437572532|
|    min|                 0|
|    max|                79|
+-------+------------------+
```

```
df.select('station_id').distinct().count()
```

```
843
```

# Spark Dataframe: conversion with Pandas

```
# conversion with Pandas
import pandas as pd

pandaDf = df.toPandas()

pandaDf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 843 entries, 0 to 842
Data columns (total 18 columns):
station_id                  843 non-null object
name                        843 non-null object
short_name                  843 non-null object
latitude                    843 non-null object
longitude                   843 non-null object
region_id                   843 non-null object
rental_methods              843 non-null object
capacity                    843 non-null object
eightd_has_key_dispenser    843 non-null object
num_bikes_available         843 non-null object
num_bikes_disabled          843 non-null object
num_docks_available         843 non-null object
num_docks_disabled          843 non-null object
is_installed                843 non-null object
is_renting                  843 non-null object
is_returning                843 non-null object
eightd_has_available_keys   843 non-null object
last_reported               843 non-null object
dtypes: object(18)
memory usage: 118.6+ KB
```

# Work with Spark SQL

```
# Play with Spark SQL
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("citibike")
```

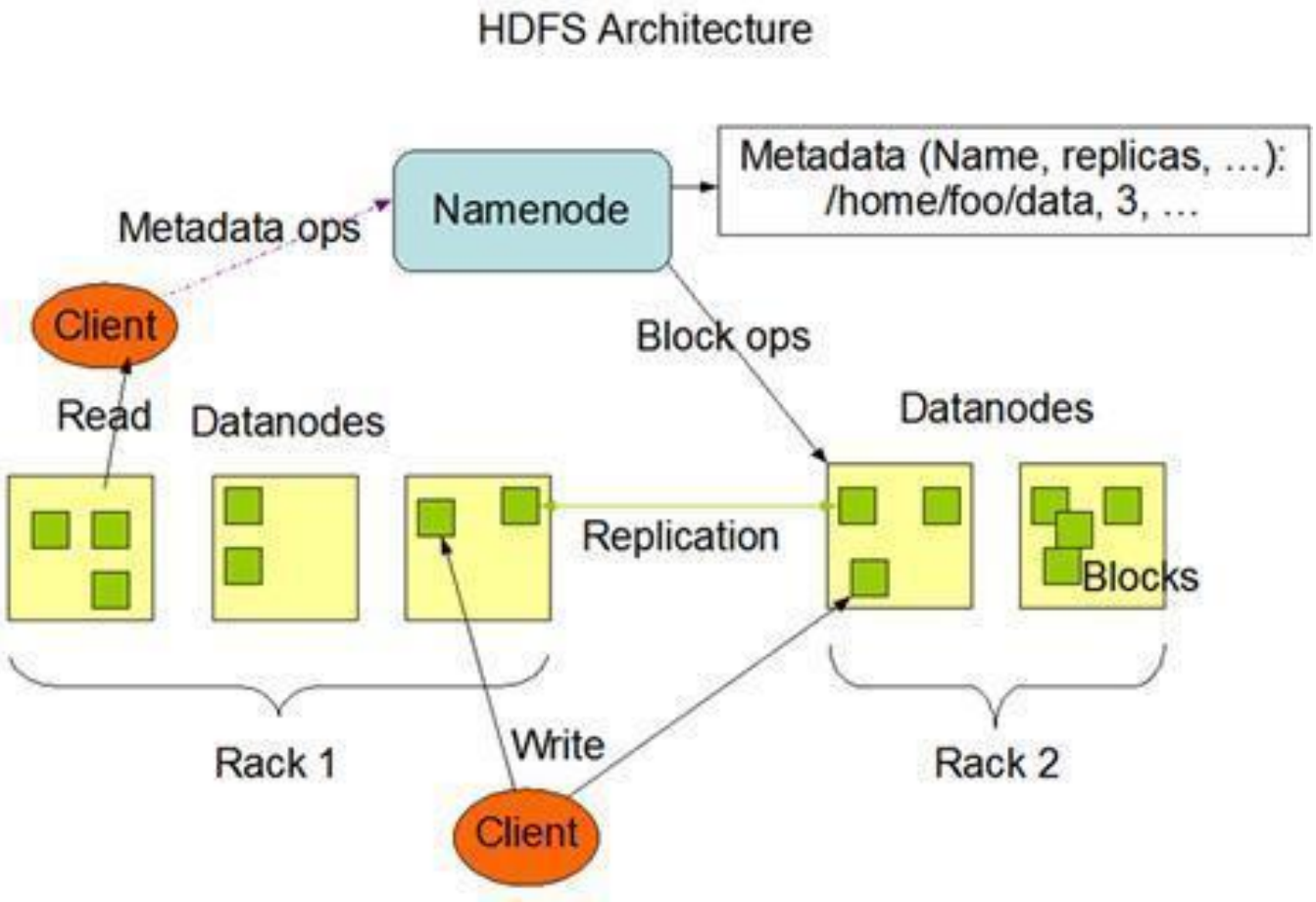
```
sqlDF = spark.sql("""
        SELECT COUNT (DISTINCT station_id)
        FROM citibike
        """)
sqlDF.show()
```

```
+--------------------------+
|count(DISTINCT station_id)|
+--------------------------+
|                       843|
+--------------------------+
```

```
# get data out of df
sqlDF.select("count(DISTINCT station_id)").collect()[0]["count(DISTINCT station_id)"]
```

```
843
```

# Hadoop metrics




https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

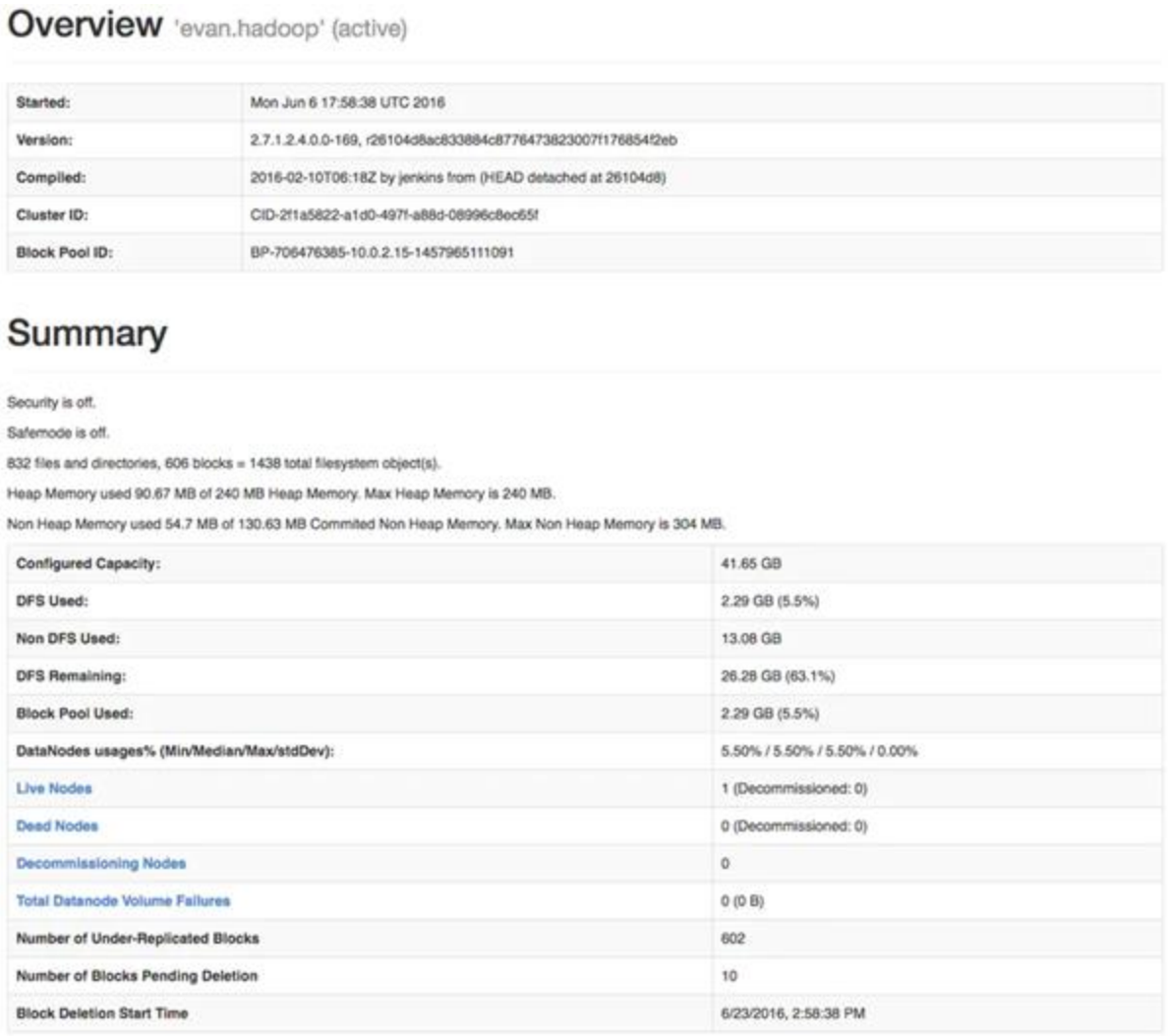
# Collecting HDFS metrics

- Collecting NameNode metrics via API
- Collecting DataNode metrics via API
- Collecting HDFS metrics via JMX

# Collecting HDFS metrics

- **NameNode HTTP API**

  The NameNode offers a summary of health and performance metrics through an easy-to-use web UI. By default, the UI is accessible via port 50070, so point a web browser at: http://<namenodehost>:50070



**Overview** 'evan.hadoop' (active)

| | |
|---|---|
| **Started:** | Mon Jun 6 17:58:38 UTC 2016 |
| **Version:** | 2.7.1.2.4.0.0-169, r26104d8ac833884c87764738230071f17685412eb |
| **Compiled:** | 2016-02-10T06:18Z by jenkins from (HEAD detached at 26104d8) |
| **Cluster ID:** | CID-2f1a5822-a1d0-497f-a88d-08996c8ec65f |
| **Block Pool ID:** | BP-706476385-10.0.2.15-1457965111091 |

**Summary**

Security is off.

Safemode is off.

832 files and directories, 606 blocks = 1438 total filesystem object(s).

Heap Memory used 90.67 MB of 240 MB Heap Memory. Max Heap Memory is 240 MB.

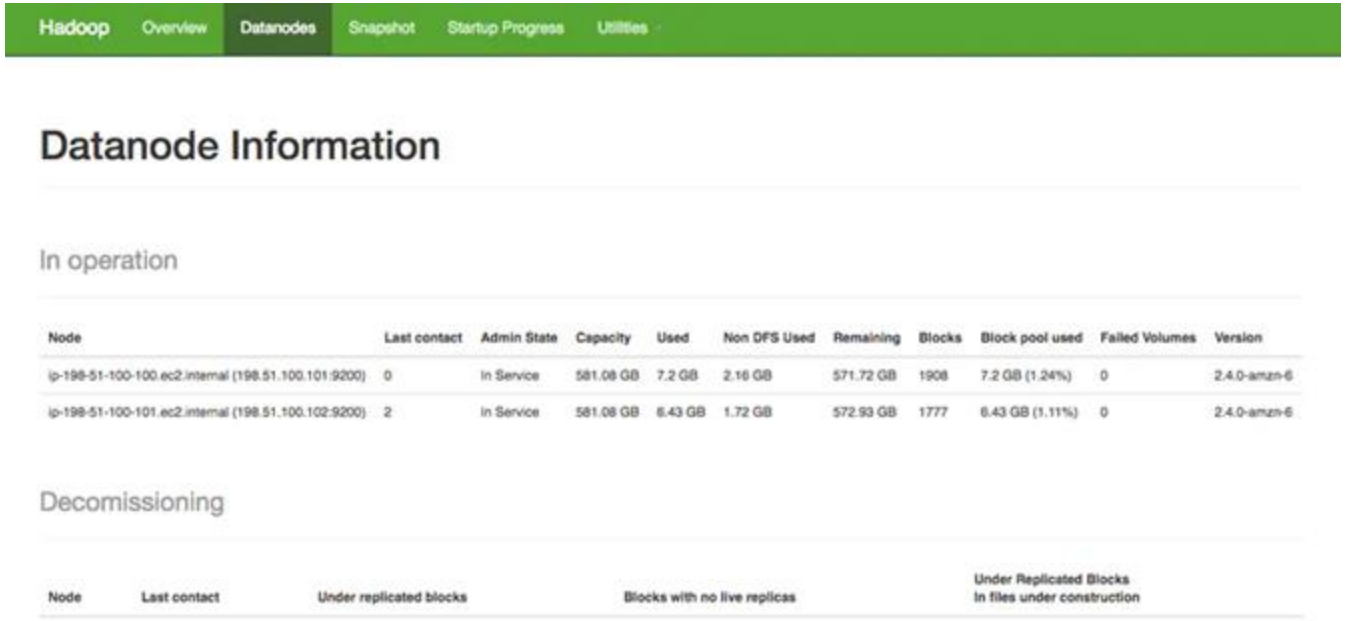
Non Heap Memory used 54.7 MB of 130.63 MB Commited Non Heap Memory. Max Non Heap Memory is 304 MB.

| | |
|---|---|
| **Configured Capacity:** | 41.65 GB |
| **DFS Used:** | 2.29 GB (5.5%) |
| **Non DFS Used:** | 13.08 GB |
| **DFS Remaining:** | 26.28 GB (63.1%) |
| **Block Pool Used:** | 2.29 GB (5.5%) |
| **DataNodes usages% (Min/Median/Max/stdDev):** | 5.50% / 5.50% / 5.50% / 0.00% |
| Live Nodes | 1 (Decommissioned: 0) |
| Dead Nodes | 0 (Decommissioned: 0) |
| Decommissioning Nodes | 0 |
| Total Datanode Volume Failures | 0 (0 B) |
| **Number of Under-Replicated Blocks** | 602 |
| **Number of Blocks Pending Deletion** | 10 |
| **Block Deletion Start Time** | 6/23/2016, 2:58:38 PM |

# Collecting HDFS metrics

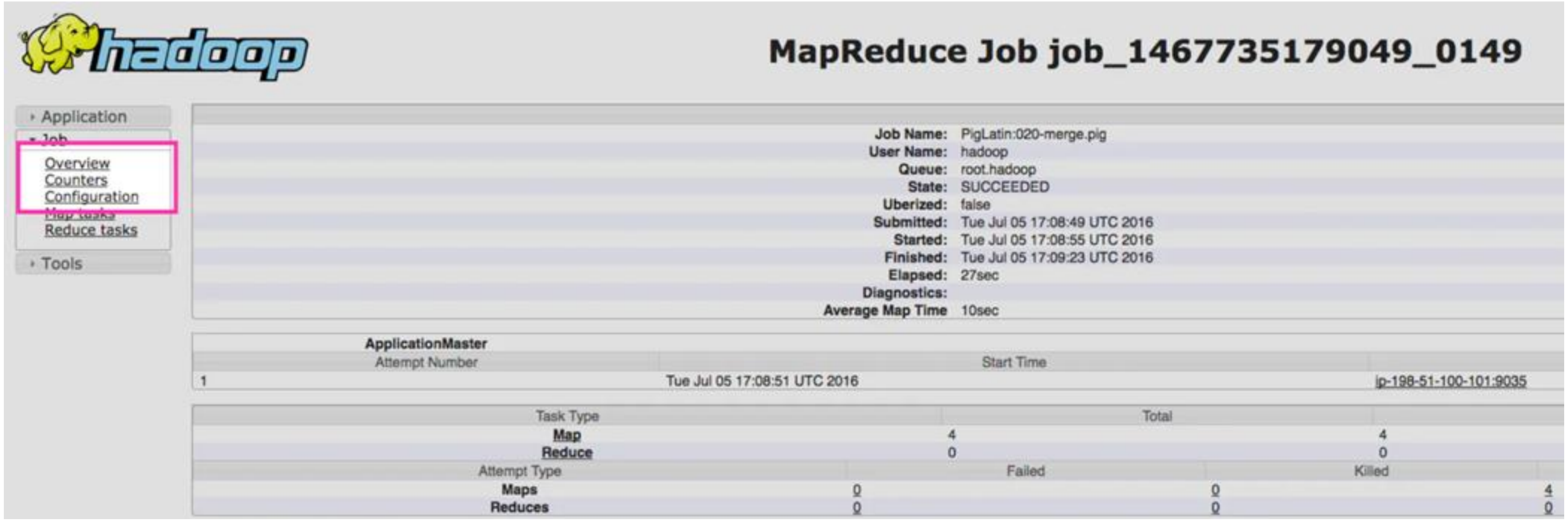
- **DataNode HTTP API**

A high-level overview of the health of your DataNodes is available in the NameNode dashboard, under the Datanodes tab (http://localhost:50070/dfshealth.html#tab-datanode)

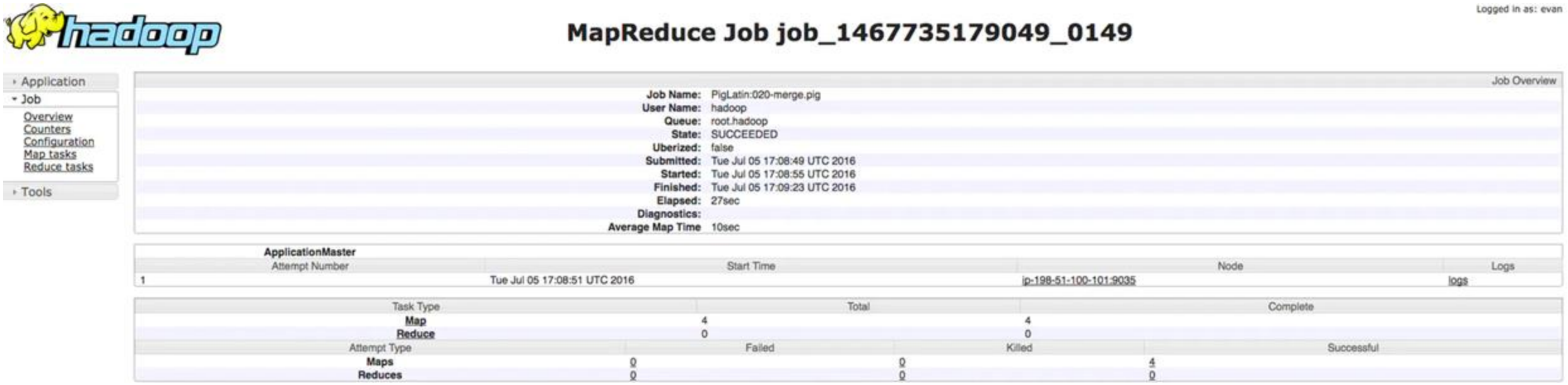


# Collecting MapReduce counters

MapReduce counters provide information on MapReduce task execution, like CPU time and memory used. They are dumped to the console when invoking Hadoop jobs from the command line, which is great for spot-checking as jobs run, but more detailed analysis requires monitoring counters over time.

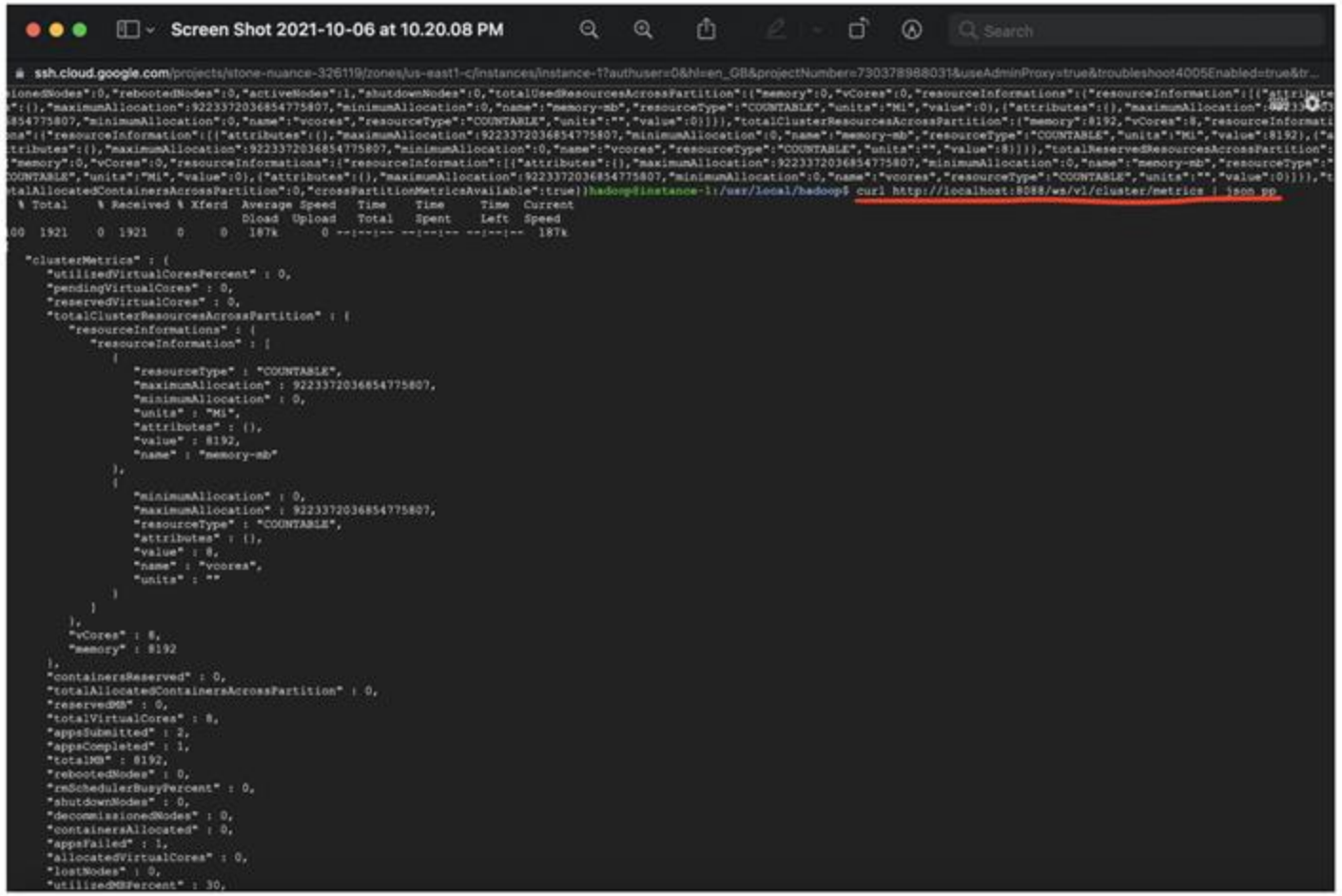


# Collecting MapReduce counters



# Collecting Hadoop YARN metrics

By default, YARN exposes all of its metrics on port 8088, via the jmx endpoint. Hitting this API endpoint on your ResourceManager gives you all of the metrics from part two of this series,



# Third-party tools

- Apache Ambari
- Cloudera Manager

# References


- https://spark.apache.org/docs/latest/sql-getting-started.html
- https://www.analyticsvidhya.com/blog/2016/10/spark-dataframe-and-operations/
- https://spark.apache.org/docs/latest/ml-guide.html
- https://towardsdatascience.com/machine-learning-with-pyspark-and-mllib-solving-a-binary-classification-problem-96396065d2aa
- https://www.datadoghq.com/blog/collecting-hadoop-metrics/#namenode-and-datanode-metrics-via-jmx
- https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-common/Metrics.html

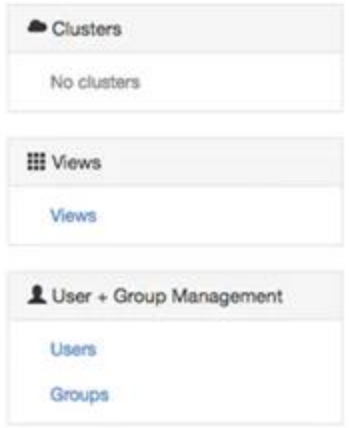
# Third-party tools

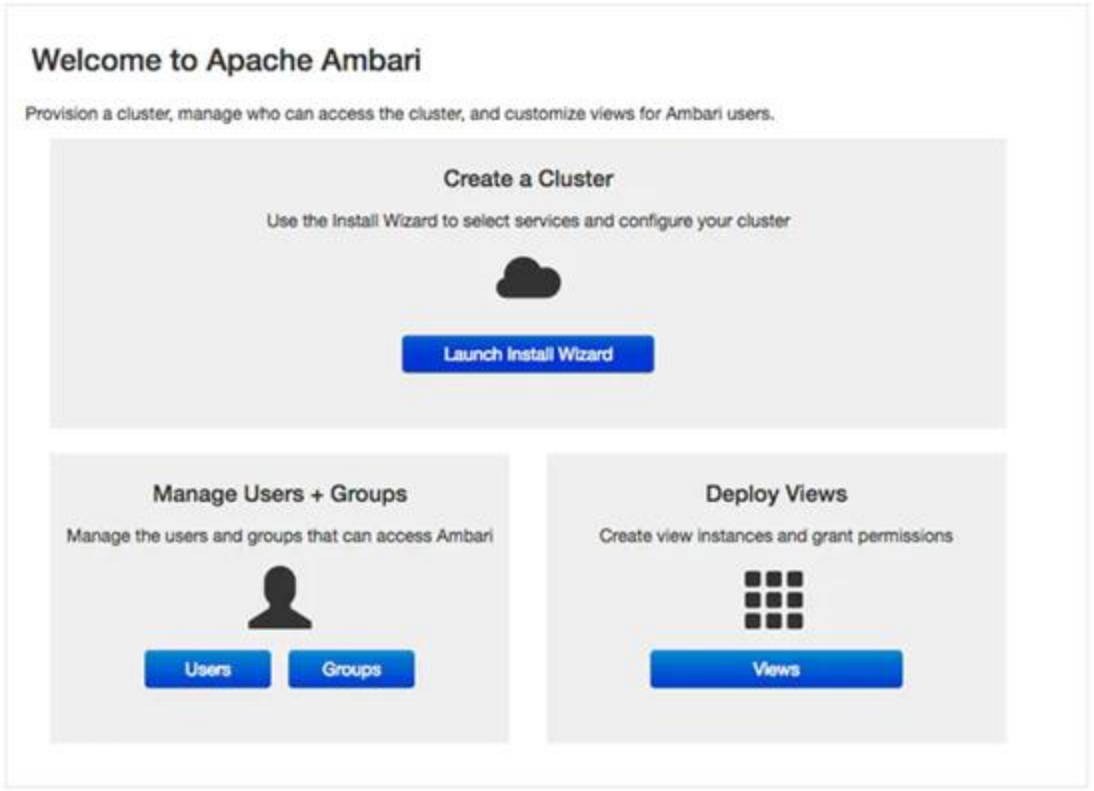
- **Apache Ambari**

ambari-server setup

service ambari-server start

point your browser to <AmbariHost>:8080 and login with the default user admin and password admin

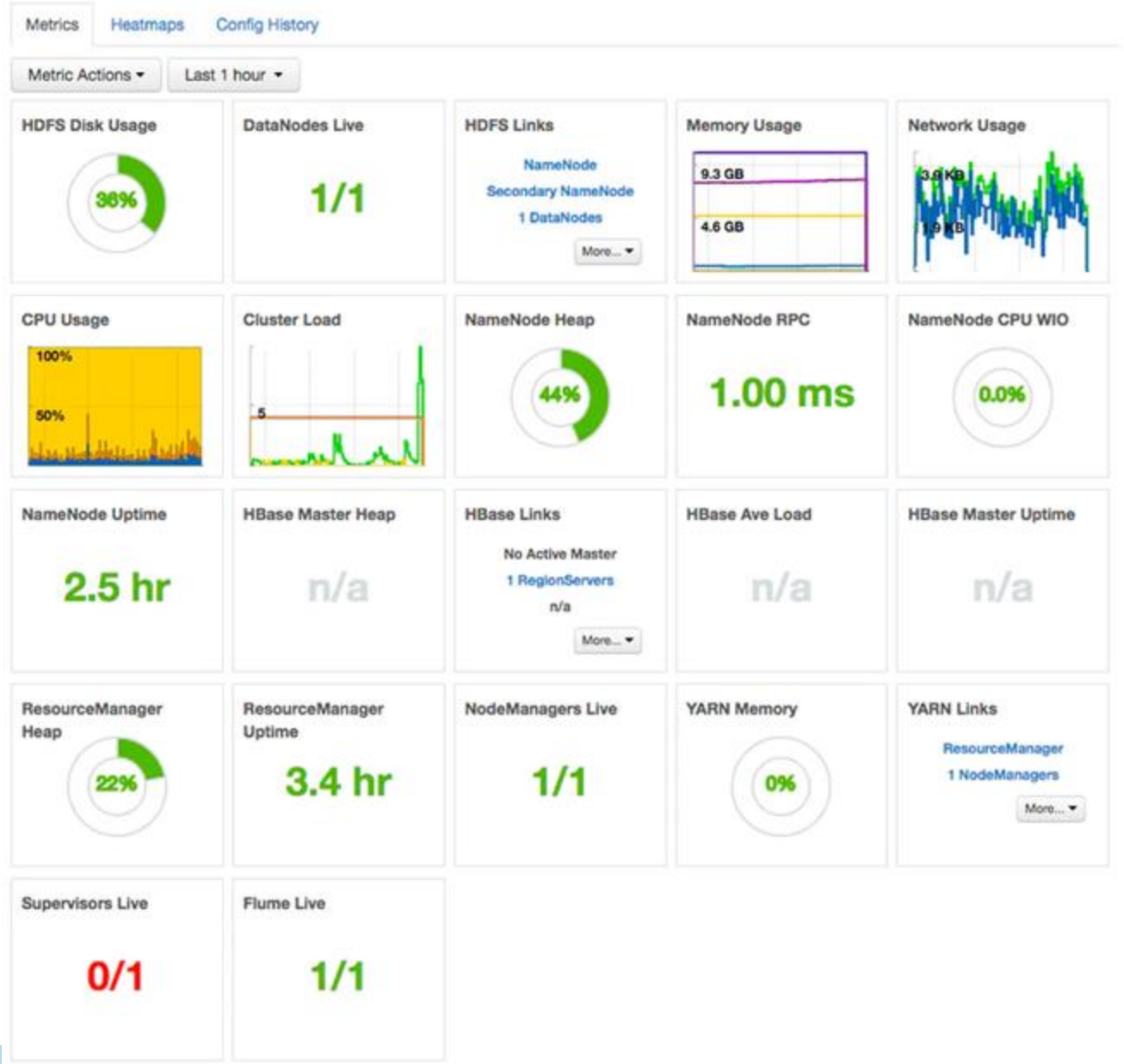


# Third-party tools

- **Apache Ambari**

Select “Launch Install Wizard”, the series of screens that follow, you will be prompted for hosts to be monitored and credentials to connect to each host in your cluster, then you’ll be prompted to configure application-specific settings.



# Third-party tools

- **Cloudera Manager**

service cloudera-scm-server start

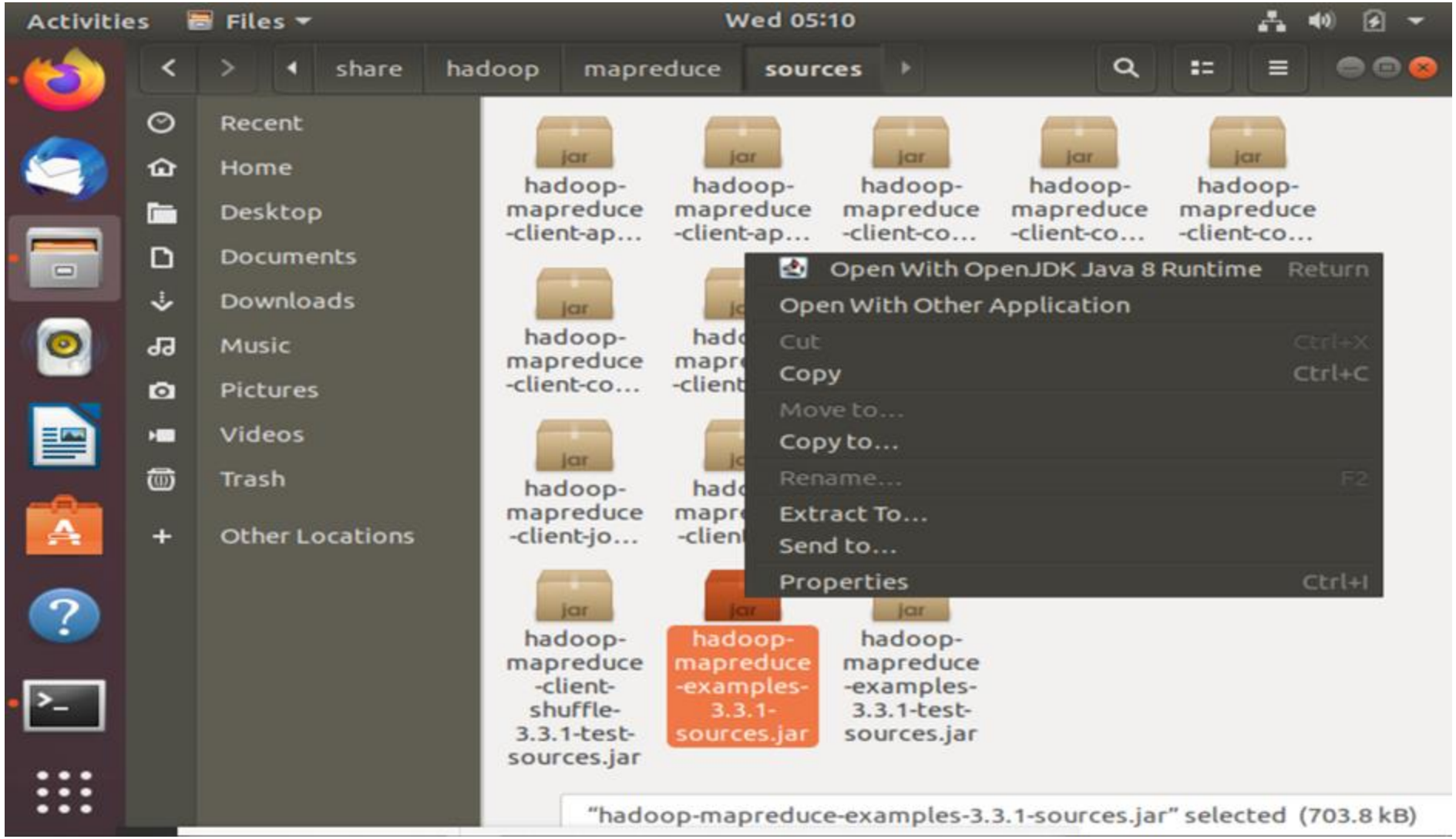
point your browser to <ClouderaHost>:7180 and login with the default user admin and password admin.

# Hadoop metrics: take wordcount as an example

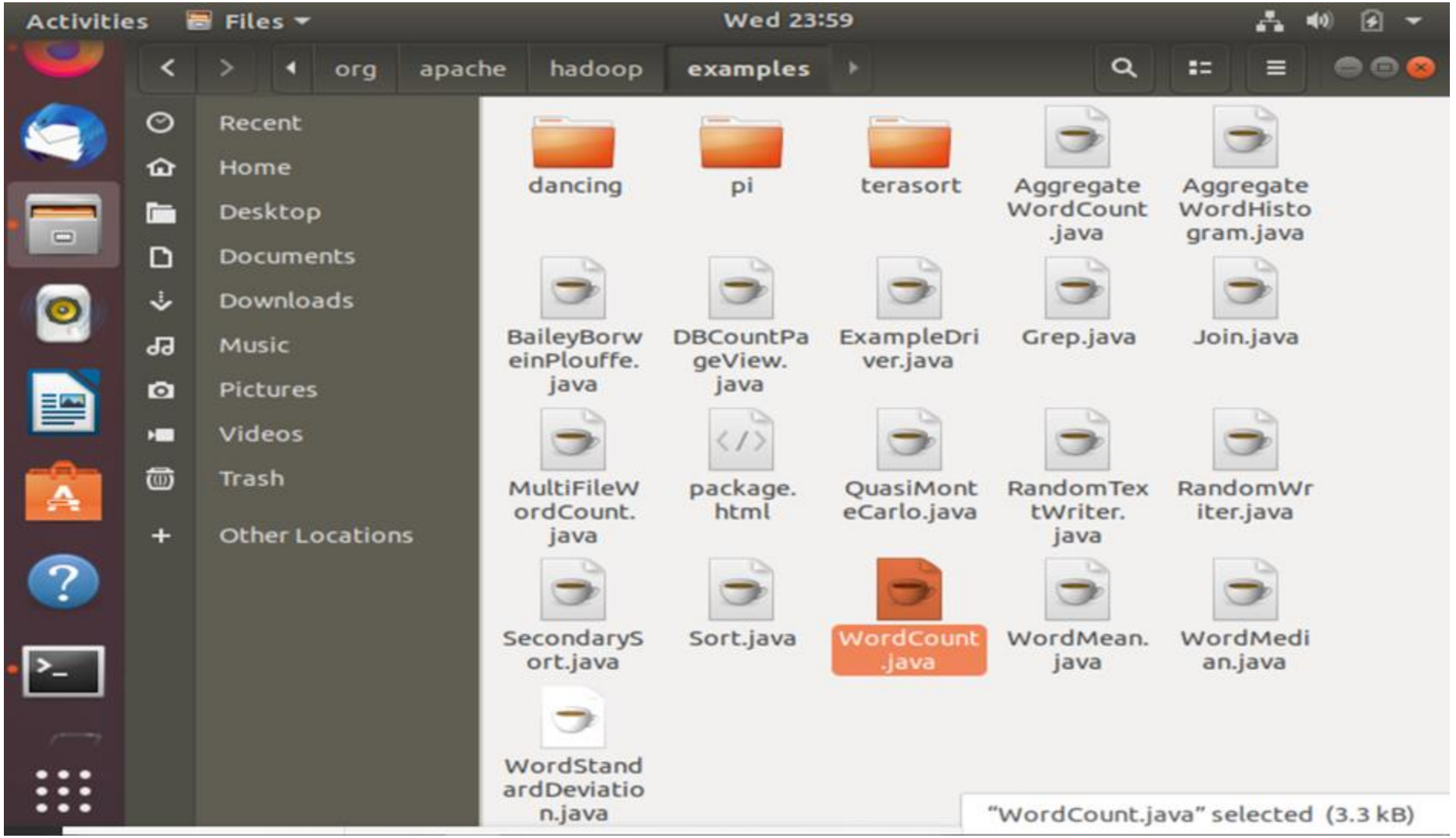
| files | content | task |
| --- | --- | --- |
| aa.txt | this is data and intelligence | Sort by alphabet, count the number of occurrences of each word in the three files. |
| bb.txt | hello everyone | |
| cc.txt | welcome | |

| input | output |
| --- | --- |
| This is data and intelligence<br><br>Hello everyone<br><br>welcome | and 1 |
| | data 1 |
| | everyone 1 |
| | hello 1 |
| | intelligence 1 |
| | is 1 |
| | this 1 |
| | welcome 1 |

# Hadoop metrics: take wordcount as an example



# Hadoop metrics: take wordcount as an example

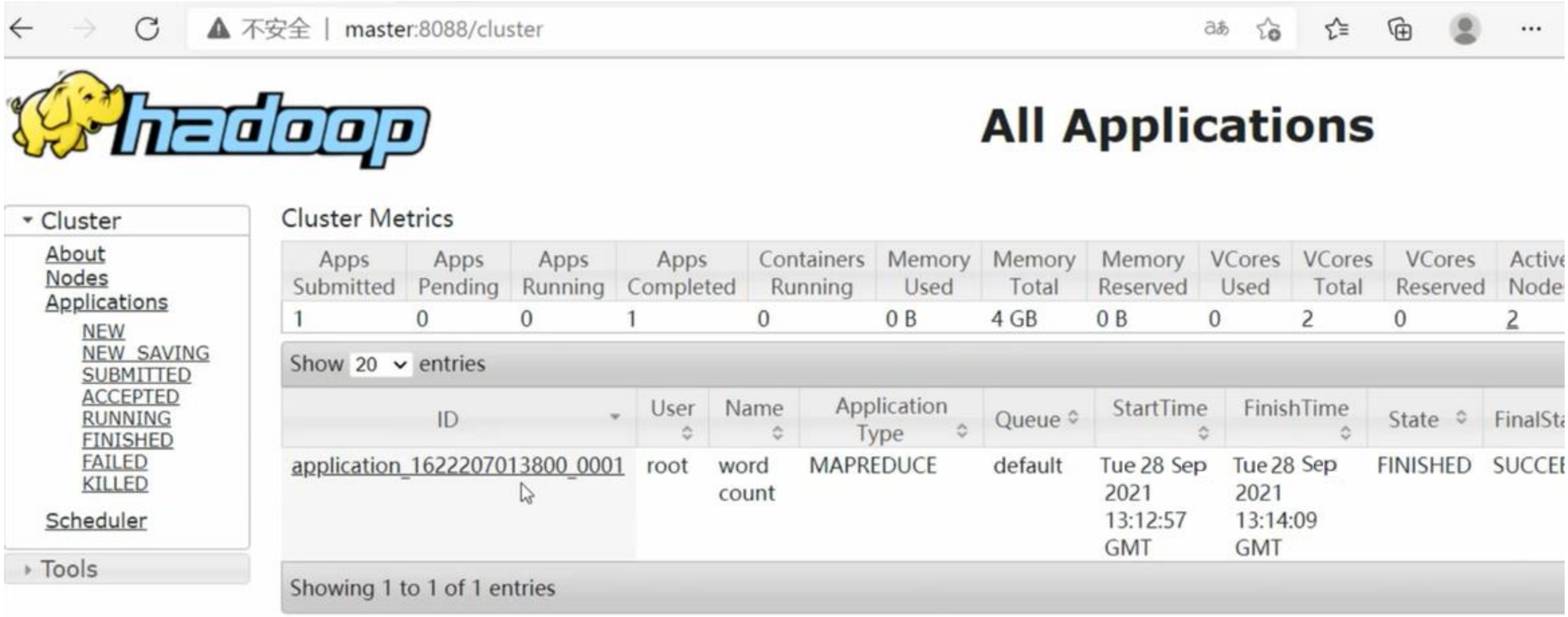


# Hadoop metrics: take wordcount as an example

The parameters after wordcount is input files except last one.
The last one “newwordcount” is output folder.

```
[root@master ~]# hadoop jar /usr/local/hadoop-2.6.5/share/hadoop/mapreduce/hadoop-mapreduce
-examples-2.6.5.jar wordcount /usr/root/txtdir/aa.txt /usr/root/txtdir/bb.txt /usr/root/txt
dir/cc.txt /usr/root/txtdir/newwordcount
21/09/28 21:12:47 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your
 platform... using builtin-java classes where applicable
21/09/28 21:12:50 INFO client.RMProxy: Connecting to ResourceManager at master/192.168.65.1
00:8032
21/09/28 21:12:54 INFO input.FileInputFormat: Total input paths to process : 3
21/09/28 21:12:55 INFO mapreduce.JobSubmitter: number of splits:3
21/09/28 21:12:56 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1622207013800
_0001
21/09/28 21:12:58 INFO impl.YarnClientImpl: Submitted application application_1622207013800
_0001
21/09/28 21:12:58 INFO mapreduce.Job: The url to track the job: http://master:8088/proxy/ap
plication_1622207013800_0001/
21/09/28 21:12:58 INFO mapreduce.Job: Running job: job_1622207013800_0001
21/09/28 21:13:23 INFO mapreduce.Job: Job job_1622207013800_0001 running in uber mode : fal
se
21/09/28 21:13:23 INFO mapreduce.Job:  map 0% reduce 0%
21/09/28 21:13:42 INFO mapreduce.Job:  map 33% reduce 0%
21/09/28 21:13:51 INFO mapreduce.Job:  map 67% reduce 0%
```

# Hadoop metrics: take wordcount as an example



# Hadoop metrics: take wordcount as an example