307401
Big Data and Data Warehouses

Running MapReduce Using MRJob Python Library

Implement MapReduce Jobs using MRJOB

Java is the standard language for running MapReduce jobs.

mrjob is a Python library developed by Yelp that lets us:

- Write MapReduce jobs entirely in Python.
- **mrjob** can run MaprReduce jobs locally, on Hadoop, or on EMR (AWS) with the same code.
- It allows us to focus on logic, not infrastructure.

Thinking in MapReduce

Core Idea

- MapReduce breaks large-scale data processing into two logical steps—Map and Reduce—each focusing on a small, local unit of data.

Mapper Logic

- "Given this one input record, what small pieces of information do I want to extract or reformat and pass on to the next step?"
- Processes one record at a time
- Emits (key, value) pairs for further grouping
- Focuses on local transformations such as filtering, parsing, or mapping

Reducer Logic

- "For this one key, I now have all its associated values from across all mappers. How do I combine them into a final result?"
- Receives all values for the same key
- Performs aggregation or summarization (for example, sum, count, or average)

What MapReduce Handles Automatically

- Distributes data across machines
- Sorts and groups intermediate pairs by key
- Feeds grouped data into reducers

Design Mindset

- Think locally first:
- Mapper: What can I compute or emit from one record?
- Reducer: How do I combine all outputs for one key?

Thinking in MapReduce

So the mapper code always looks something like this:

```
def map(key, value):
    # process one record
    emit(new_key, new_value)
```

While the reducer operates on one key and all its associated values at a time.

When you write a reducer, you think like this:

```
def reduce(key, values):
    # values is a list or iterator of all values for this key
    result = aggregate(values)
    emit(key, result)
```

MRJob example: Words Count

Python MapReduce code to count the words in our file Input.txt

```
from mrjob.job import MRJob

class MRWordCount(MRJob):
    def mapper(self, _, line):
        for word in line.split():
            yield word.lower(), 1

    def reducer(self, word, counts):
        yield word, sum(counts)

if __name__ == '__main__':
    MRWordCount.run()
```

```
input.txt file
hello world bye world
hello hadoop mapreduce
world
```

Part	Description
from mrjob.job import MRJob	Imports the MRJob base class used to define MapReduce jobs.
class MRWordCount(MRJob):	Defines a new job called MRWordCount that inherits from MRJob.
def mapper(self, _, line):	The **mapper** function runs on each line of input text.
for word in line.split(): yield word, 1	Splits the line into words and emits each word paired with the number 1.
def reducer(self, word, counts):	The **reducer** function takes all counts for each word.
yield word, sum(counts)	Adds up all counts for a word and emits the total.

Step-by-Step: Who Does What

lifecycle of a key–value pair:

- **Mapper emits (key, value):**
  - This happens in the *mapper process* (a JVM container on a node).
  - These pairs are first kept in memory in a buffer.
  - When the buffer fills, Hadoop **spills** the data to **local disk** in sorted order by key.
- **Mapper output → local disk:**
  - Each mapper writes one or more **spill files**.
  - When the mapper finishes, it **merges** all spill files into one final sorted output file (still on the mapper's local disk).
- **Shuffle and transfer phase:**
  - The **reducer processes** contact each mapper node (over HTTP) to fetch the relevant partitions of the mapper outputs.
  - This is managed by the **Hadoop framework**, not your code.
  - Data is transferred over the network: each reducer fetches the partitions of intermediate data corresponding to its assigned key range.
- **Sort and merge on the reducer side:**
  - Each reducer merges all fetched mapper partitions, sorts by key again (if needed), and groups values for each key.
  - Then it calls your reducer function for each group.
- **Reducer output:**
  - The reducer writes final results to **HDFS** (Hadoop Distributed File System).

Run Code Locally

1) Download and install Python version **3.10 or 3.11** (they support MRJob) – Look for Windows installer (64-bit)
https://www.python.org/downloads/release/python-3119/

2) Open the command line terminal and install mrjob on your local computer:
```
pip install mrjob
```

3) Download words_count.py and input.txt files from BlackBoard.

4) Open the command line terminal and type:
```
python words_count.py input.txt
```

You should see:
```
"bye"     1
"hadoop" 1
"hello"   2
"mapreduce" 1
"world"   3
```

This verifies that the logic works before distributing the job.

Understanding What Happened

- Hadoop divided your file into chunks (input splits).
- Each **mapper** processed one split, generating intermediate (word, 1) pairs.
- Hadoop grouped identical keys (all “hello”s together).
- The **reducer** summed values for each key.
- This is the essence of **MapReduce** — distributed computation through key-value pair processing.

Movie Ratings Count Example

User ID	Movie ID	Rating	Time Stamp
0	50	5	881250949
0	172	5	881250949
0	133	1	881250949
196	242	3	881250949
186	302	3	891717742
22	377	1	878887116
244	51	2	880606923
166	346	1	886397596
298	474	4	884182806
115	265	2	881171488
253	465	5	891628467
305	451	3	886324817

Using MRStep (for multi-step jobs)

When you have multiple map/reduce phases, use MRStep, for example, counting word frequency then sorting by frequency.

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class RatingsBreakdown(MRJob):
    def steps(self):
        return [MRStep(mapper=self.mapper_get_ratings,
                       reducer=self.reducer_count_ratings),
            MRStep(reducer=self.reducer_sort_by_count)]

    def mapper_get_ratings(self, _, line):
        # Input: userID, movieID, rating, timestamp (tab-separated)
        try:
            userID, movieID, rating, timestamp = line.strip().split('\t')
            yield rating, 1
        except ValueError:
            pass  # skip malformed lines

    def reducer_count_ratings(self, rating, counts):
        # Count how many times each rating appears
        yield None, (sum(counts), rating)

    def reducer_sort_by_count(self, _, count_rating_pairs):
        # Sort by count (descending)
        sorted_pairs = sorted(count_rating_pairs, reverse=True)
        for count, rating in sorted_pairs:
            yield rating, count

if __name__ == '__main__':
    RatingsBreakdown.run()
```

MRStep in mrjob:

- MRStep defines one MapReduce phase consisting of an optional mapper, combiner, and reducer.
- It allows chaining multiple MapReduce steps within a single MRJob class.
- The steps() method returns a list of MRStep objects executed sequentially.
- Output from one step automatically becomes input for the next step.
- Each MRStep runs as a separate MapReduce job under the hood.
- Common use: perform an initial computation (e.g., counting) in the first step, then process or sort those results in a later step.

In the example:

- Step 1 mapper emits (rating, 1); reducer counts occurrences and emits (None, (count, rating)).
- Step 2 reducer receives all (count, rating) pairs, sorts them, and emits (rating, count).
- Using MRStep avoids writing multiple separate jobs and enables building pipelines inside one mrjob script.

Step	Function	Input Example	Operation / Logic	Output Example
1. Mapper	`mapper_get_ratings(_, line)`	Each line of the file: `1 31 2.5 1260759144`	Splits each line by tab into `userID`, `movieID`, `rating`, `timestamp`. Emits `(rating, 1)` for each line.	`(2.5, 1)`
2. Shuffle & Sort (automatic)	*(Hadoop/MRJob built-in)*	Mapper outputs: `(2.5,1), (3.0,1), (2.5,1)`	Groups values by key (`rating`) and sorts keys in ascending order before passing to reducers.	`2.5 → [1,1], 3.0 → [1]`
3. Reducer #1	`reducer_count_ratings(rating, counts)`	`rating = "2.5", counts = [1,1]`	Sums the counts for each rating. Emits all results **under a single key None so they go to the same reducer in the next step**.	`(None, (2, "2.5"))`
4. Shuffle #2 (automatic)	*(Hadoop/MRJob built-in)*	All pairs `(None, (count, rating))`	**Since all share the same key None, they are sent together to one reducer.**	`[(2, "2.5"), (1, "3.0"), ...]`
5. Reducer #2	`reducer_sort_by_count(_, count_rating_pairs)`	`[(2, "2.5"), (1, "3.0"), (5, "4.0")]`	Sorts all pairs by count in descending order. Emits `(rating, count)` for each sorted pair.	`"4.0" → 5, "2.5" → 2, "3.0" → 1`
6. Final Output	—	—	Displays ratings sorted by how often they appear.	```

4.0 5

2.5 2

3.0 1

Running MapReduce on Hadoop

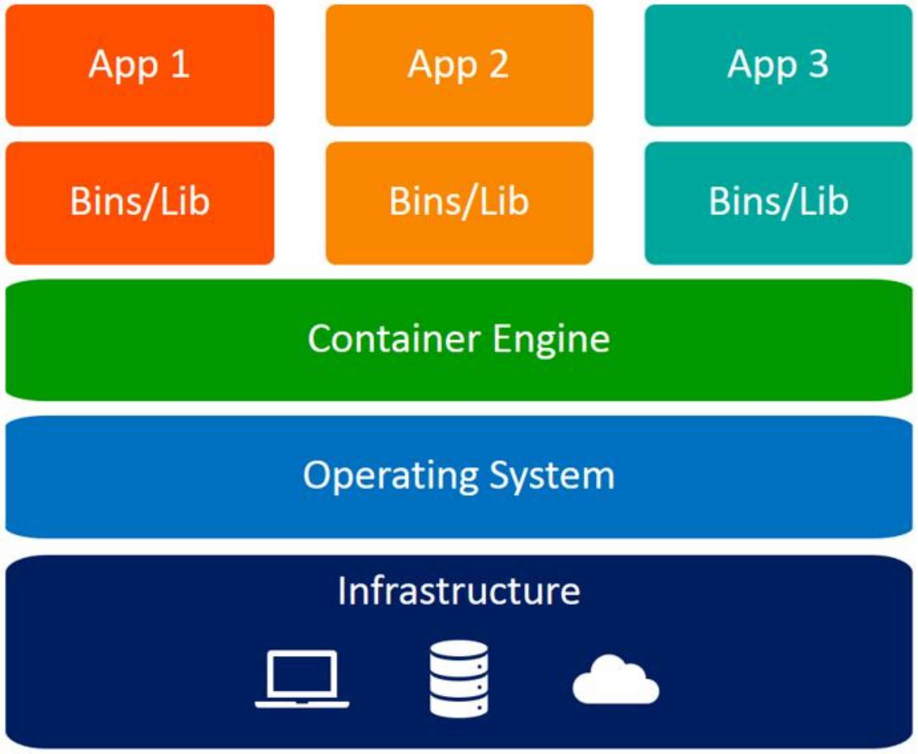
Introduction to Docker and Docker Compose

What is Docker

- Docker is a containerization platform that allows you to package an application together with all its dependencies into a single portable unit called a container.
- Containers ensure that the application runs the same way on any system, regardless of the underlying operating system or configuration.
- Unlike virtual machines, Docker containers share the host system's operating system kernel, making them lightweight, efficient, and fast to start.
- Analogy: A Docker container is like a sealed lab box that includes everything your program needs to run.

Why We Use Docker in This Lab

- To create a ready-to-use Hadoop environment without complex installation steps
- To prevent version or dependency conflicts between systems
- To make the lab easily reproducible on student laptops or classroom machines
- To simulate a cluster setup using a single machine



Containers

Installing and Configuring Docker

Step 1: Install Docker Desktop

1. Go to https://www.docker.com/products/docker-desktop
2. Download and install **Docker Desktop for Windows**.
3. During setup, **enable the WSL 2 backend**.
4. Restart your computer.
5. Verify Docker is running — the **whale icon** should appear in the system tray.

Step 2: Adjust Docker Resources

1. Open **Docker Desktop → Settings → Resources**.
2. Allocate:
   - CPUs: 2
   - Memory: 4–6 GB
   - Swap: 1–2 GB
3. Click **Apply & Restart**.

Proper resource allocation is critical. Hadoop requires enough memory for its daemons (NameNode, DataNode, ResourceManager, etc.) to start successfully.

1) Download and Run the Hadoop Container

Open the command line and the following commands:

a. Pull Hadoop container from the Internet (Docker Hub) - open a Windows command line terminal and type the command below:

docker pull msfasha/hadoop-petra:latest

b. Run the container on you machine - open a Windows command line terminal and type the command below:

docker run -it -p 9870:9870 -p 8088:8088 --name hadoop-lab msfasha/hadoop-petra

c. We can check the running Hadoop environment using Web UI:

HDFS UI: http://localhost:9870

YARN UI: http://localhost:8088

d. Enter the container shell - open a Windows command line terminal and type the command below:

docker exec -it hadoop-lab bash

Basic test commands you can run inside Hadoop container:

```
# check Hadoop daemons
jps
# list files in HDFS
hdfs dfs -ls /
# check running containers or YARN jobs
yarn application -list
```

2) Copy Code File and Data File Into the Container

a. Download (words_count.py and inputs.txt) from Black Board.

b. Copy files from your machine into the docker container - open a Windows command line terminal and type the command below:

docker cp words_count.py hadoop-lab:/opt/hadoop/

docker cp input.txt hadoop-lab:/opt/hadoop/

c. Enter the docker container – type the command below inside Windows command line terminal:

docker exec -it hadoop-lab bash

c. Copy data file from the container into Hadoop HDFS – type the command below in Hadoop container command line:

hdfs dfs -mkdir /input

hdfs dfs -put /opt/hadoop/input.txt /input/

3) Run the MapReduce Job in Hadoop

a. Run the Python MapReduce Program on Hadoop -

python3 /opt/hadoop/words_count.py -r hadoop hdfs:///input/input.txt -o hdfs:///output

Explanation:

- -r hadoop: Run using Hadoop’s MapReduce engine.
- hdfs:///input/input.txt: Input file in HDFS.
- -o hdfs:///output_mrjob: Output folder in HDFS.
- Monitor progress in YARN’s web interface: http://localhost:8088

4) Check the Results

a. When the job completes, check the output in HDFS:

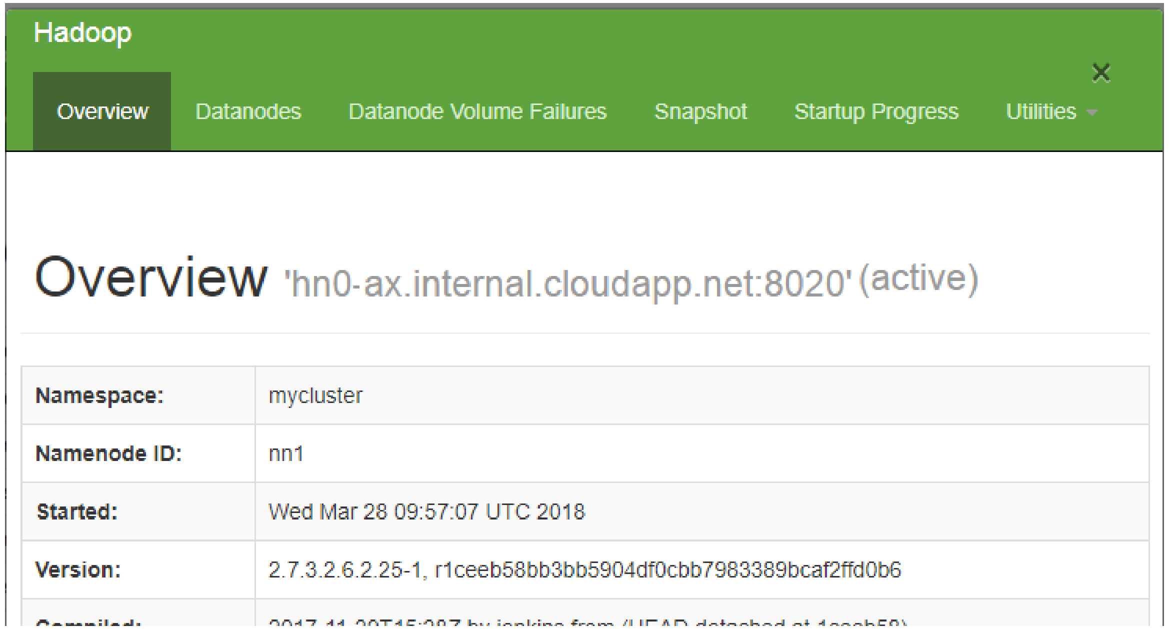
```
hdfs dfs -cat /output/part-00000
```

Expected output:

```
"bye"    1
"hadoop" 1
"hello"  2
"mapreduce" 1
"world"  3
```

HDFS Web UI - YARN UI: http://localhost:9870

Use Case:
The HDFS Web UI provides a visual interface for monitoring and managing the Hadoop Distributed File System. It Helps administrators monitor cluster health, storage status, and performance from a centralized web interface



Main Components:

- **Overview:** Shows active NameNode details (cluster name, version, start time, status).
- **Datanodes:** Lists all connected DataNodes, their capacity, usage, and health.
- **Datanode Volume Failures:** Displays any disk or node-level failures.
- **Snapshot:** Manages and reviews HDFS snapshots for data recovery.
- **Startup Progress:** Tracks NameNode initialization steps during startup.
- **Utilities:** Offers tools for diagnostics, logs, and configuration checks.

Application Manager - YARN UI: http://localhost:8088

Use Case: Administrators and developers use this interface to monitor, manage, and troubleshoot Hadoop applications in real time.

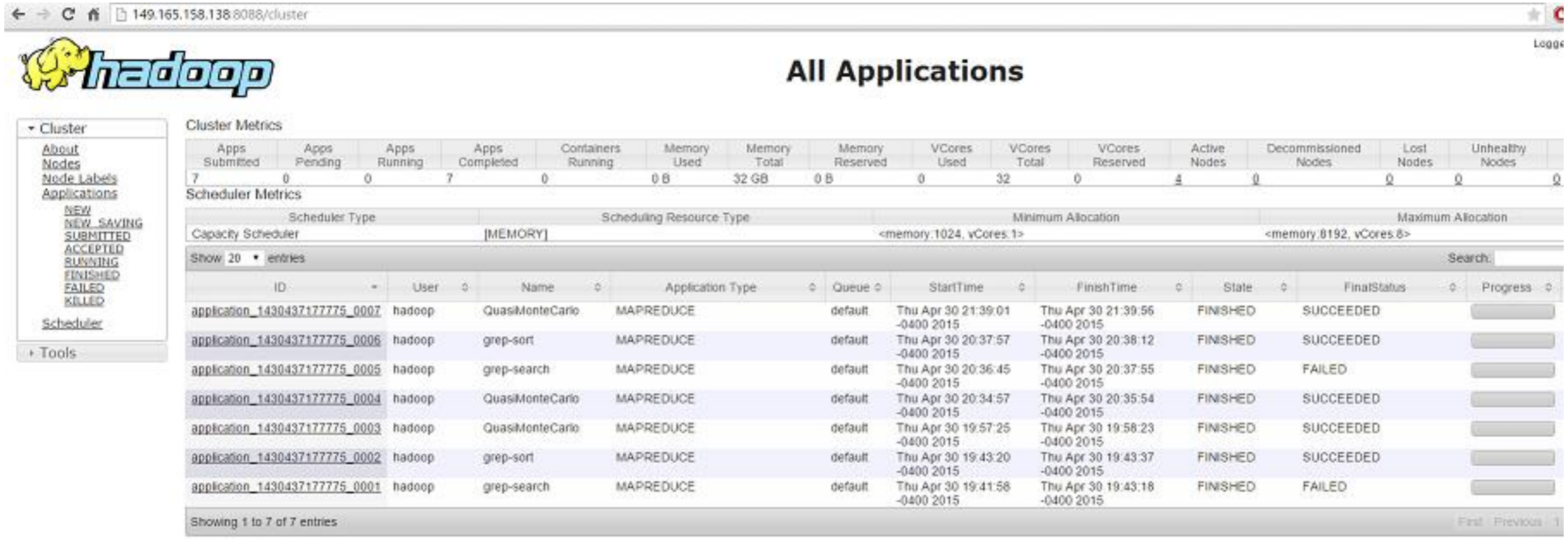
Key Sections:

Cluster Metrics: Displays overall cluster performance statistics such as:

- **Apps Submitted / Running / Pending / Completed:** Shows the total number of jobs and their states.
- **Memory & vCores:** Indicates total and used resources (e.g., Memory Used: 32 GB, vCores Used: 32).
- **Node Information:** Active, Decommissioned, Lost, or Unhealthy Nodes.

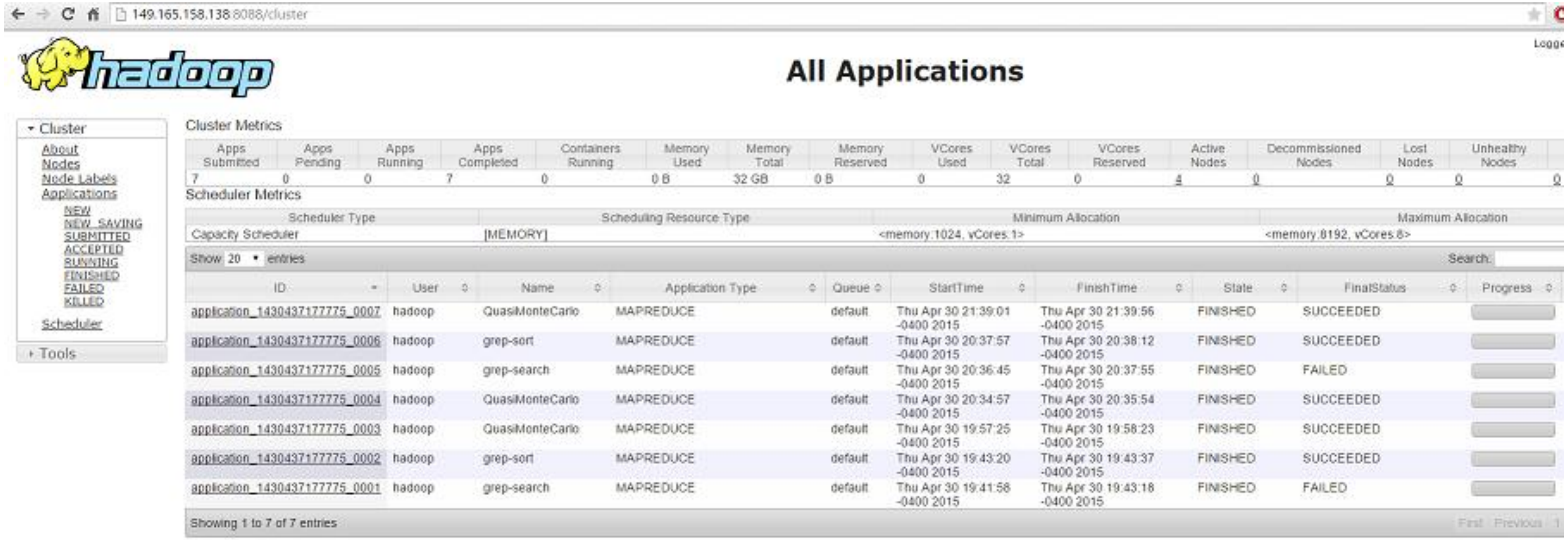
Scheduler Metrics:

- Shows configuration of the Capacity Scheduler (resource allocation policy).
- Lists resource constraints like memory and vCore allocation ranges.



Applications Table: Lists individual Hadoop jobs with details such as:

- **Application ID:** Unique identifier (e.g., application_1430473177775_0002)
- **User:** The user who submitted the job (e.g., hadoop)
- **Application Type:** Type of job (MAPREDUCE)
- **Queue:** Scheduling queue used (default)
- **Start and Finish Times:** Execution duration
- **State / Final Status:** Job completion outcome (e.g., FINISHED / SUCCEEDED or FAILED)
- **Progress Bar:** Visual indication of job progress.



Key Takeaways

- Hadoop allows distributed processing using MapReduce.
- Docker provides an easy way to run Hadoop without complex installation.
- HDFS stores data across multiple nodes; YARN manages job execution.
- mrjob makes writing Python MapReduce programs straightforward.
- Hadoop web interfaces offer valuable visualization and debugging tools.