PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

SAAD DAHLEB BLIDA 01 UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

MASTER'S INTELLIGENT SYSTEMS ENGINEERING

**DATA WAREHOUSE AND BIG DATA**

REPORT

# Implementation of a Hadoop Multi-Node Cluster and Spark Framework for Image Compression

Implemented and written by

Abdelatif Mekri

Halima Nfidsa

Imad Eddine Boukader

Nahla Yasmine Mihoubi

Academic year : 2024-2025

# Table of content

Table of content..... 1
Introduction..... 3
Project Objectives..... 3
Technologies Used..... 3
Fundamentals..... 4
Image Compression..... 4
Image Compression Techniques..... 4
Lossless Compression..... 4
Lossy Compression..... 5
Comparison of Lossless and Lossy Compression..... 5
How Hadoop Work for Image Compression..... 6
Storing Images in HDFS..... 6
Image Compression Using MapReduce..... 6
How Spark and Hadoop Work Together for Image Compression..... 7
Role of Hadoop (HDFS)..... 8
1. Distributed Storage:..... 8
2. Data Availability:..... 8
3. Input/Output for Compression:..... 9
Role of Spark..... 9
1. Reading Images:..... 9
2. In-Memory Processing:..... 9
3. Parallel Compression:..... 9
4. Saving Results:..... 9
Benefits of Combining Hadoop and Spark..... 9
Project Impact..... 10
Installation and Configuration of the Environment..... 10
A- Virtualisation Environment..... 10
B- Setting up the Machines..... 11
1- Operating System Selection and Installation..... 11
2- Hadoop Installation and Configuration..... 11
2-1 Installing JAVA..... 11
2-2 Installing Hadoop..... 12
2-3 Making ‘slave’ machines..... 14
2-4 Configuring SSH for Passwordless Login..... 14
2-5 Getting IP addresses and renaming hosts..... 15
2-6 setting up the hosts files..... 16
2-7 configuring hadoop on master and streaming the configuration to the slaves...

17
2-8 launching hadoop....20
3- Spark installation and configuration....22
3-1 Installing JAVA....22
3-2 Download Apache Spark....22
3-3 Installing Spark....22
3-4 Configure Environment Variables....23
3-5 Update Workers file....23
3-6 update workers with the new config....24
3-7 launching Spark....25
Code and running....26
I. First approach : Hadoop & spark....26
Code....26
A- Installing and importing the necessary libraries....26
B- Implementing the solution....26
Starting hadoop and spark....28
Check the HDFS....28
Upload the images to the designated folder in HDFS....29
Job submission....29
Running and waiting for job compilation....29
Retrieving results....30
II. Second approach : Hadoop (Map-Reduce)....31
Code....31
Compiling the java files....32
Starting hadoop....32
Check the HDFS....32
Upload the images to the designated folder in HDFS....32
Job submission....33
Running and waiting for job compilation....33
Downloading the output files....33
Fix the output file to see the results....34
Results....34
Conclusion and Discussion....35
Notes....36

## Keywords :

Hadoop ,spark , image compression , mapreduce ,distributed storage, HDFS

# Introduction

Image compression is an essential technique in modern data processing, especially for large-scale multimedia datasets. It allows for the efficient reduction of file sizes while preserving the visual quality of images. In this project, we explore the application of **Big Data technologies**—**Hadoop** and **Spark**—to implement a parallelized image compression pipeline using the **JPEG** format. These formats are widely recognized for their ability to compress images efficiently, maintaining high image quality at lower file sizes. The primary goal of this project is to demonstrate the effectiveness of Hadoop and Spark in optimizing the compression process for large image datasets, focusing on distributed and in-memory processing capabilities, respectively. By leveraging these two technologies, we aim to highlight the scalability, speed, and efficiency improvements in handling and compressing vast amounts of image data.

## Project Objectives

**Setup and Configuration of Hadoop and Spark Environments**: Install and configure both Hadoop and Spark frameworks, ensuring they are ready for processing large image datasets in a distributed and in-memory fashion.

1. **Development and Implementation of Image Compression Algorithms**: Implement efficient image compression algorithms using the JPEG and JPEG2000 formats, ensuring minimal loss of visual quality while achieving significant file size reduction.
2. **Integration of Hadoop and Spark for Enhanced Performance**: Utilize Hadoop's distributed storage system and MapReduce framework alongside Spark's in-memory processing engine to parallelize the image compression workflow, optimizing resource usage and reducing processing times.
3. **Comparative Analysis of Hadoop vs. Spark**: Analyze the performance of Hadoop and Spark in the context of image compression, comparing aspects such as processing time, scalability, resource utilization, and overall efficiency when handling large datasets.

## Technologies Used

- **Hadoop**

  Hadoop  is an open-source Big Data framework designed to efficiently store and process large datasets across distributed clusters. In this project, Hadoop's components were used to facilitate the storage and parallel processing of images:
  - **HDFS (Hadoop Distributed File System)**: HDFS provides a robust and fault-tolerant distributed storage system, enabling the storage of massive

image files across multiple nodes. This makes it well-suited for large-scale image data processing, ensuring that even when some nodes fail, the data remains accessible and can be recovered.

- **MapReduce**: Hadoop's MapReduce framework was leveraged to divide the image compression task into smaller sub-tasks, which can be processed independently and in parallel. This parallelization speeds up the overall compression process and ensures that the system can handle larger datasets more efficiently.

- **Spark**  :

  Apache Spark is a high-performance, in-memory data processing engine known for its speed and scalability. It processes data much faster than Hadoop by storing intermediate results in memory rather than writing them to disk. In this project, Spark was used to enhance the image compression process:

  - **RDDs (Resilient Distributed Datasets)**: RDDs are the core data abstraction in Spark. They allow for the distribution of image data across multiple nodes and enable parallel processing of image compression tasks. RDDs provide fault tolerance and the ability to perform transformations and actions on data in parallel.
  - **Parallel Image Compression**: By utilizing Spark's powerful transformation capabilities (such as map(), filter(), and reduce()), the compression process was parallelized across all nodes in the cluster. This approach significantly accelerates the overall compression time, making it possible to handle large datasets efficiently and reduce computational overhead.

# Fundamentals

## Image Compression

Image compression is the process of reducing the size of an image file while maintaining its quality as much as possible. It is essential for efficient storage, transmission, and processing of images, especially in applications like cloud storage, web applications, and big data analytics.

### Image Compression Techniques

#### Lossless Compression

Lossless compression methods retain the original image quality by encoding the data efficiently without losing any information. Common techniques include:

- **Run-Length Encoding (RLE)**
- **Huffman Coding**
- **Lempel-Ziv-Welch (LZW)**
- **PNG and GIF formats**

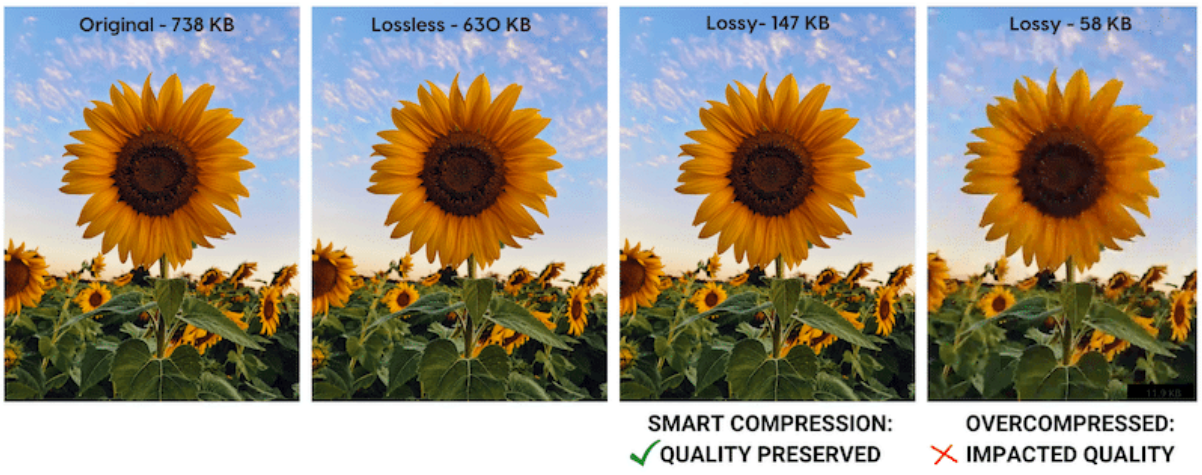
### Lossy Compression

Lossy compression techniques reduce file size significantly by discarding some image data, leading to a slight loss in quality. Common methods include:

- **Discrete Cosine Transform (DCT)** (used in JPEG)
- **Wavelet Transform** (used in JPEG2000)
- **Quantization and Downsampling**

### Comparison of Lossless and Lossy Compression

| Type of Compression | Lossless Compression | Lossy Compression |
|---|---|---|
| Principle | Algorithms used to precisely reconstruct original data from compressed data. | Uses approximate estimations to represent content. |
| Usage | Text, programs, images, and sound. | Images, audio, and video. |
| Applications | RAW, BMP, PNG, WAV, FLAC, ALAC, et | JPEG, GIF, MP3, MP4, OGG, H.264, MKV, etc. |
| Storage Accuracy | High (original content preserved). | Reliable but with reduced quality (data discarded). |



# How Hadoop Work for Image Compression

## Storing Images in HDFS

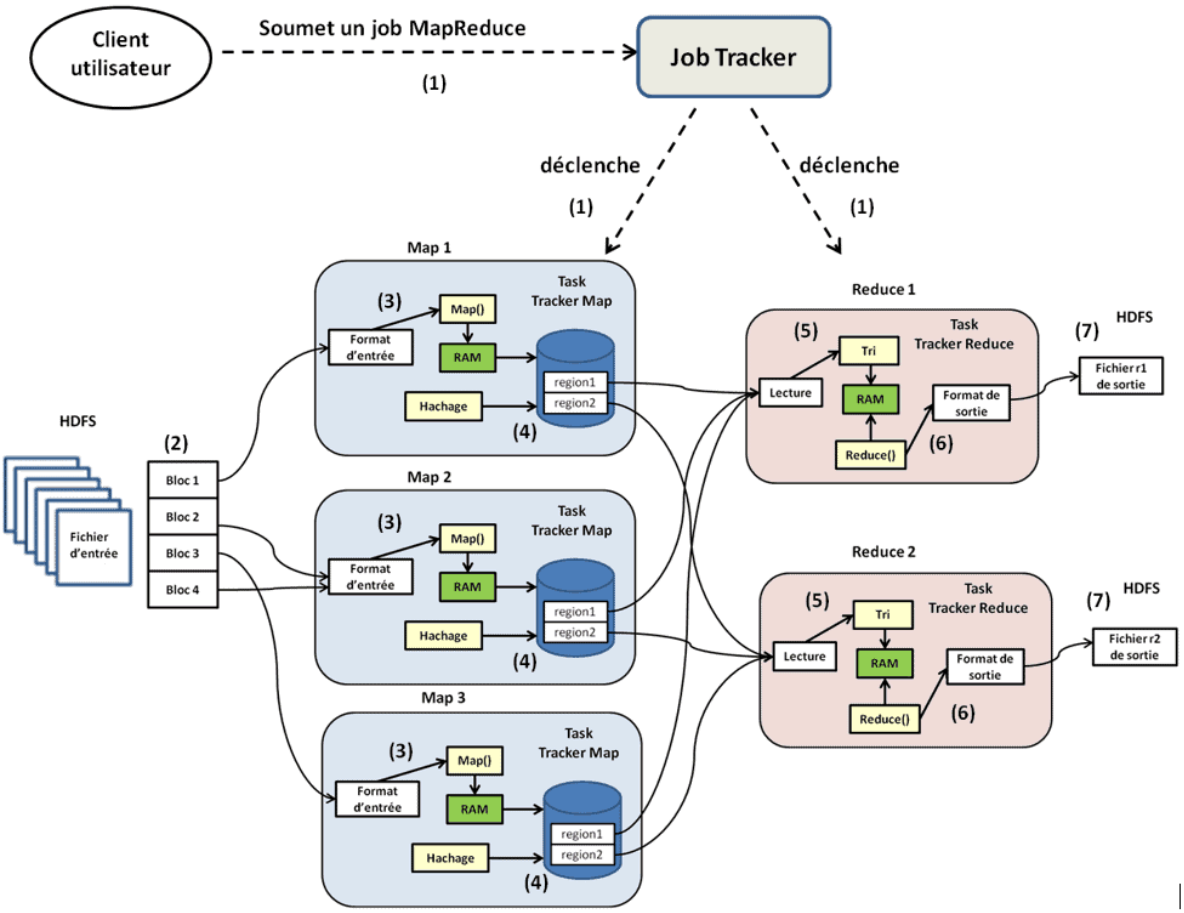
Images (JPEG files) are stored in Hadoop Distributed File System (HDFS), which breaks them into blocks and distributes them across the cluster.


A **traditional storage system**, such as a local file system (e.g., NTFS, Ext4), operates on a single machine and is limited by the system's hardware capacity. It provides fast access to files but lacks built-in fault tolerance and scalability. In contrast, the **Hadoop Distributed File System (HDFS)** is designed for distributed environments, storing data across multiple nodes to handle large-scale datasets efficiently. HDFS follows a **block-based architecture**, where files are split into large blocks (typically 128MB) and replicated across different nodes to ensure fault tolerance and high availability. Unlike traditional storage, HDFS is optimized for **sequential reads and parallel processing**, making it ideal for big data applications. Additionally, while normal file systems allow frequent updates and modifications, HDFS follows a **write-once, read-many** approach, which enhances consistency and throughput in large-scale data processing frameworks like **MapReduce and Spark**. Its distributed nature allows for horizontal scalability, ensuring that storage and processing capabilities grow seamlessly with the addition of new nodes.

### Image Compression Using MapReduce

Hadoop's MapReduce framework can be used to process images in parallel across the cluster.The process involves:

**Map Phase:** Each node processes a subset of images (or parts of images) to apply compression algorithms.
**Reduce Phase:** The compressed images are collected and stored back in HDFS.




**to summarize we have 4 Steps for JPEG Compression in Hadoop**

**Step 1:** Load images into HDFS.

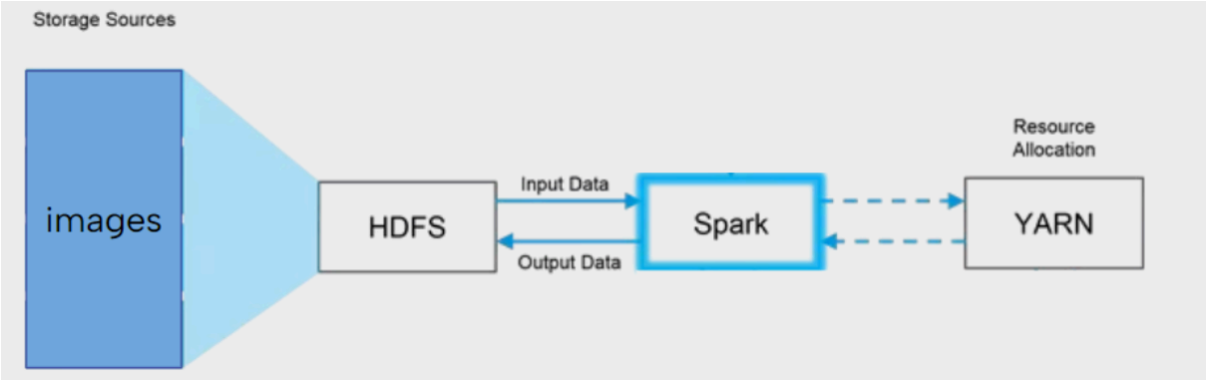
**Step 2:** Write a MapReduce job where:

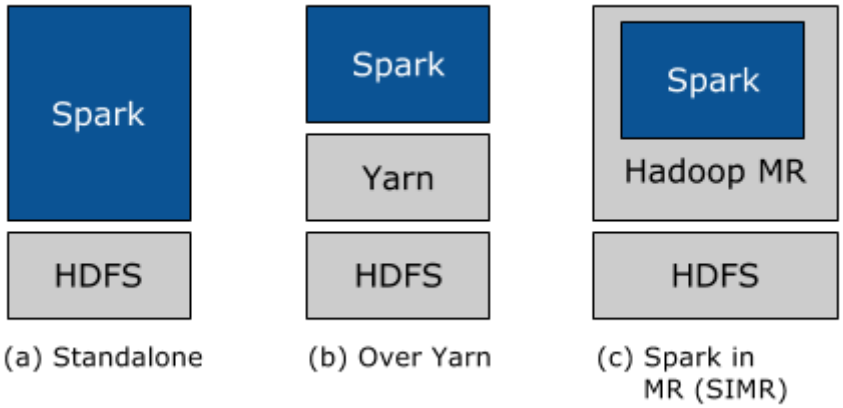
The Mapper reads the images, applies compression (e.g., reducing quality or resizing), and outputs the compressed images.

The Reducer collects the compressed images and writes them back to HDFS.

## How Spark and Hadoop Work Together for Image Compression

When combining Spark and Hadoop, their respective strengths Hadoop's distributed storage (HDFS) and Spark's in-memory processing are leveraged to build an efficient and scalable pipeline for image compression.




There are three ways to deploy Spark with Hadoop: **Standalone deployment**, where Spark runs alongside Hadoop MapReduce with manually allocated resources; **Hadoop YARN deployment**, which allows Spark to run on YARN without additional installation, integrating seamlessly into the Hadoop ecosystem; and **Spark In MapReduce (SIMR)**, which enables users without YARN to launch Spark jobs within MapReduce for quick experimentation. We chose the **Standalone deployment** because of its simplicity and ease of use, making it the preferred choice for many Hadoop users.

## Role of Hadoop (HDFS)

Hadoop Distributed File System (HDFS) serves as the backbone for data storage. Here's how it contributes:

1. Distributed Storage:
    - Images are divided into blocks and distributed across a cluster of machines for redundancy and parallel access.
    - This ensures fault tolerance and scalability for large datasets.
2. Data Availability:
    - Spark reads the images directly from HDFS, leveraging its parallel access capabilities to process data faster.

3. Input/Output for Compression:
    - HDFS acts as both the input source for uncompressed images and the output destination for compressed images.

### Role of Spark

Apache Spark handles the actual image compression with its powerful distributed computation framework:

1. Reading Images:
    - Spark reads the image files from HDFS into its distributed data structures, such as RDDs (Resilient Distributed Datasets) or DataFrames.
2. In-Memory Processing:
    - Unlike Hadoop's MapReduce, which writes intermediate results to disk, Spark performs all operations in memory, making it much faster.
    - For image compression, Spark processes multiple images simultaneously by partitioning the data across the cluster.
3. Parallel Compression:
    - Spark executes a map function across the RDD partitions, where each partition processes a subset of the images.
    - Compression is applied using libraries like Pillow or OpenCV, converting images to the desired format (e.g., JPEG).
4. Saving Results:
    - The compressed images are written back to HDFS, preserving the benefits of distributed storage.

## Benefits of Combining Hadoop and Spark

1. **Scalability:**
    - HDFS can handle petabytes of data, while Spark processes it efficiently in parallel.
2. **Speed:**
    - Spark's in-memory processing makes it faster than Hadoop's disk-based MapReduce.
3. **Fault Tolerance:**
    - Hadoop ensures data redundancy in HDFS, while Spark automatically handles task failures.
4. **Ease of Integration:**
    - Spark can directly access data from HDFS without requiring additional setup.

# Project Impact

By combining the strengths of **Hadoop** and **Spark**, this project demonstrates how distributed and in-memory processing can revolutionize the way large-scale image datasets are handled. Hadoop's ability to store vast amounts of data across multiple nodes and Spark's capacity to process data quickly in memory enables a highly efficient image compression pipeline. The comparison between Hadoop and Spark will provide insights into which technology is more suitable for different types of image processing tasks, ultimately helping to guide future decisions for large-scale data processing applications.

This work not only showcases the capabilities of Hadoop and Spark but also highlights the broader impact of Big Data technologies in the field of multimedia data processing, with potential applications in cloud storage, streaming services, and content delivery networks.

# Installation and Configuration of the Environment

## A- Virtualisation Environment

For this project, the environment was initially set up using Oracle Virtual Machine (VirtualBox). However,  VMware Workstation was chosen as the preferred virtualization solution due to its advanced features and better performance for multi-cluster setups. A total of five virtual machines (VMs) were configured, with one serving as the *master* node and the remaining four acting as *slave* nodes. This setup allowed for the distributed processing and testing of the Hadoop and Spark frameworks in a controlled and scalable environment.

**Why VMware Workstation Was Chosen Over Oracle Virtual Machine**

VMware Workstation offered several advantages over Oracle Virtual Machine, making it a better fit for this project:

1. Enhanced Performance: VMware provides better resource allocation and utilization, ensuring smoother operation of multiple VMs simultaneously.
2. Superior Networking Features: VMware's advanced networking capabilities simplified the configuration of communication between the master and slave nodes.
3. Snapshot and Cloning Functionality: VMware's snapshot feature allowed for easy backup and restoration of VM states, while cloning enabled quick replication of slave nodes.
4. Stability and Compatibility: VMware Workstation demonstrated greater stability and compatibility with the Hadoop and Spark frameworks, reducing setup and runtime issues.

5. User-Friendly Interface: VMware's intuitive interface streamlined the management of multiple VMs, saving time during configuration and deployment.

# B- Setting up the Machines

## 1- Operating System Selection and Installation

ubuntu 20.04 LTS was chosen as the operating system for all virtual machines due to its stability, long-term support (LTS), and compatibility with the project requirements. Ubuntu 20.04 provided seamless support for Java 8 installation, which was essential for running Hadoop and Spark. Additionally, its robust package management system and extensive community support made it an ideal choice for setting up and maintaining the multi-cluster environment.

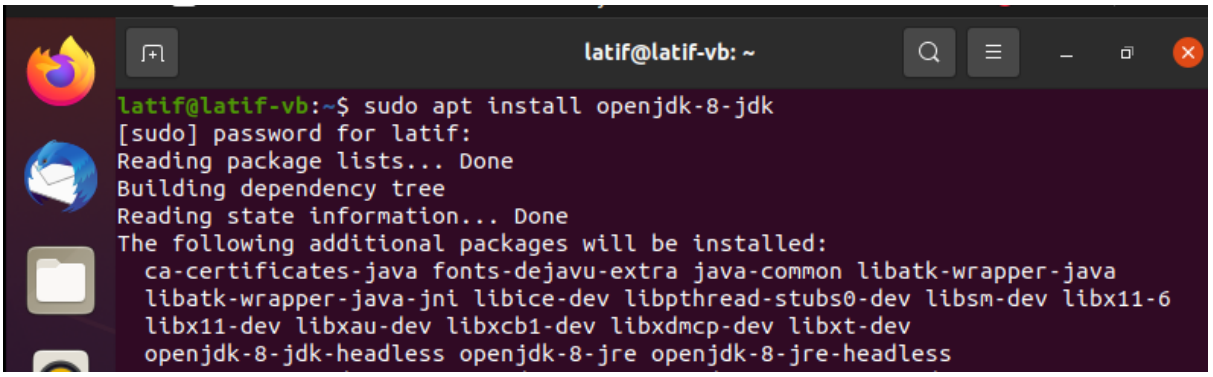
## 2- Hadoop Installation and Configuration

To set up the Hadoop framework, the following steps were performed on the master and slave nodes. The installation process was carried out on Ubuntu 20.04 LTS, ensuring compatibility with Java 8, which is a prerequisite for Hadoop.

### 2-1 Installing JAVA

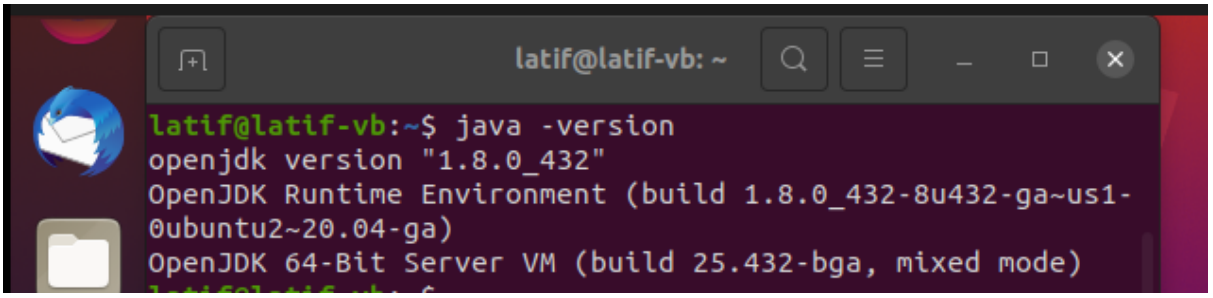
Since Hadoop requires Java to run, OpenJDK 8 was installed on the master node using the following command:

```
sudo apt install openjdk-8-jdk
```




Check the installed version of java with :

```
Java -version
```




2-2 Installing Hadoop

Open a browser on your system and navigate to the official Apache Hadoop website:

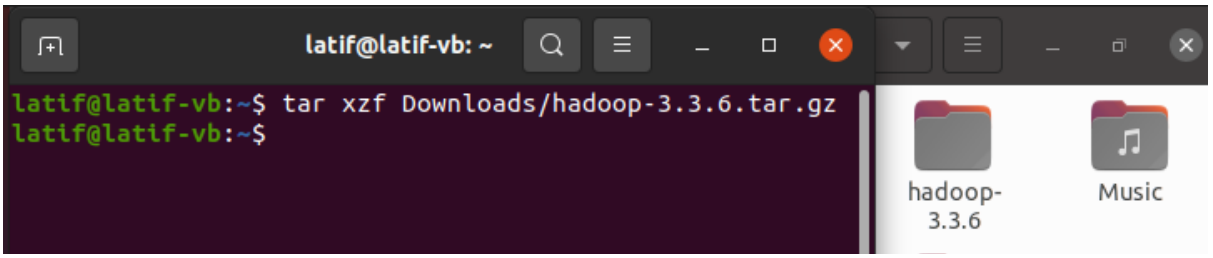
- https://hadoop.apache.org/releases.html

Select the latest stable release (in our case : Hadoop 3.3.6).
Download the **binary release** (.tar.gz file) by clicking the appropriate link for your system.

- **Extract the File**

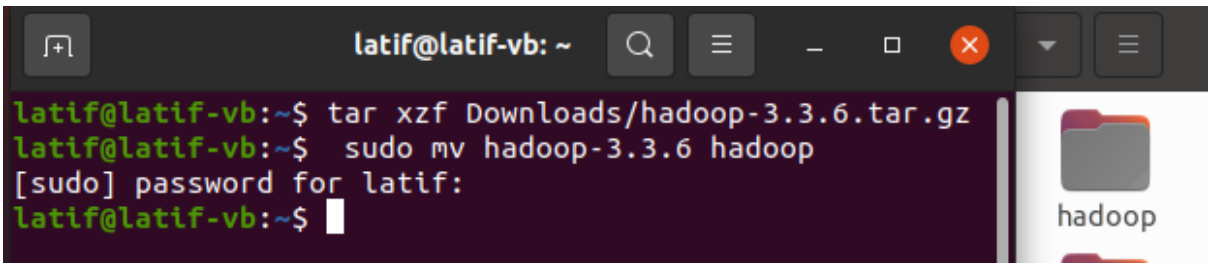
After downloading, extract the Hadoop .tar.gz , using the commande

*tar xzf Downloads/hadoop-3.2.1.tar.gz*




- **Rename the extracted folder**
  Renaming it to something simpler or more convenient (e.g., hadoop), using the commande :

  *mv hadoop-3.3.6 hadoop*




- **Start to configure the Java path:**
  on Hadoop's virtual environment , access it using the commande :

  *sudo nano ~/hadoop/etc/hadoop/hadoop-env.sh*

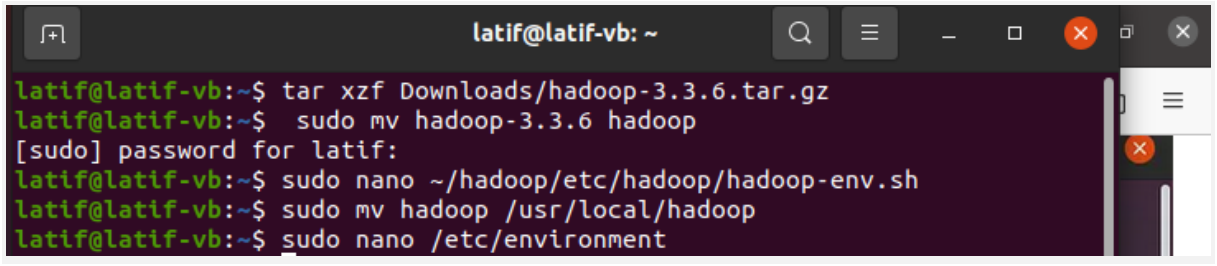
  Add the following line to the file for it to access the java path

*export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/*


- **Move the Folder to an Appropriate Location**

Move the renamed folder to /usr/local for an easy access later on , using :

*sudo mv hadoop /usr/local/hadoop*




- **set up hadoop path**

  on machine's environment, open the environment file with:

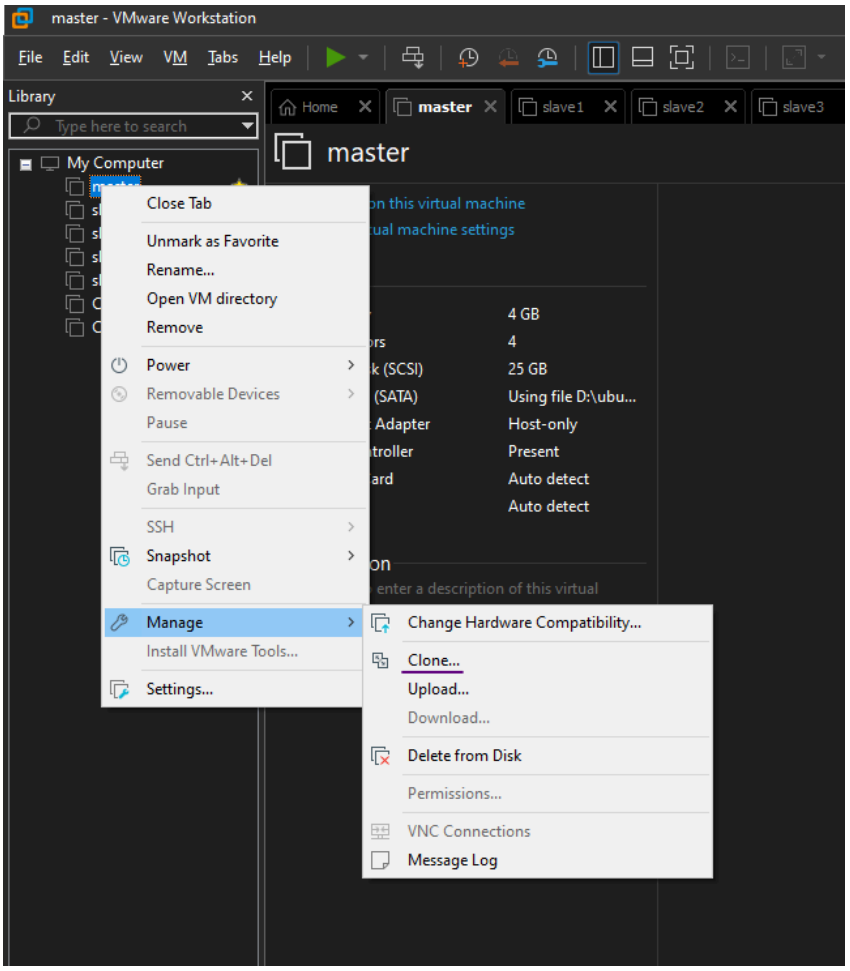
  *sudo nano /etc/environment*

Add the following lines

*'PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/usr/local/hadoop/bin:/usr/local/hadoop/sbin"*
*JAVA_HOME="/usr/lib/jvm/java-8-openjdk-amd64/jre" '*

## 2-3 Making ‘slave’ machines

In the context of setting up a Hadoop cluster, configuring the "slave" machines (or more appropriately termed "worker" nodes) is an essential part of creating the distributed system. These nodes handle the actual data storage and processing in Hadoop.




## 2-4 Configuring SSH for Passwordless Login

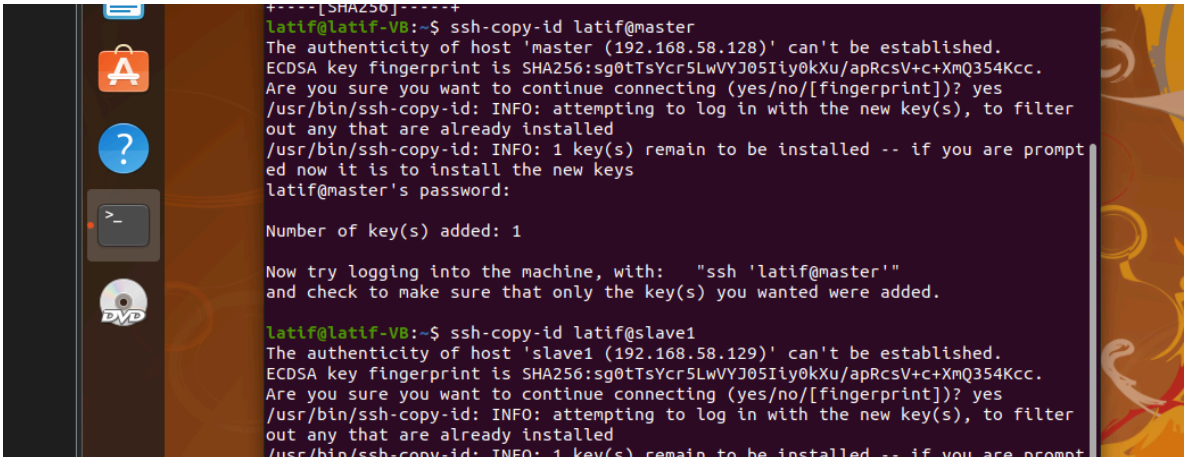
To allow the nodes in a Hadoop cluster to communicate seamlessly, we need to configure SSH for passwordless login. This ensures that the master node can execute commands on worker nodes without requiring a password

- **Generate an SSH Key Pair on the Master Node**
  On the master node, generate an SSH key pair :

  ```
  ssh-keygen -t rsa -P ""
  ```


- **Copy the Public Key to All Nodes**

Use the ssh-copy-id command to copy the master node's public key to itself and all worker nodes.

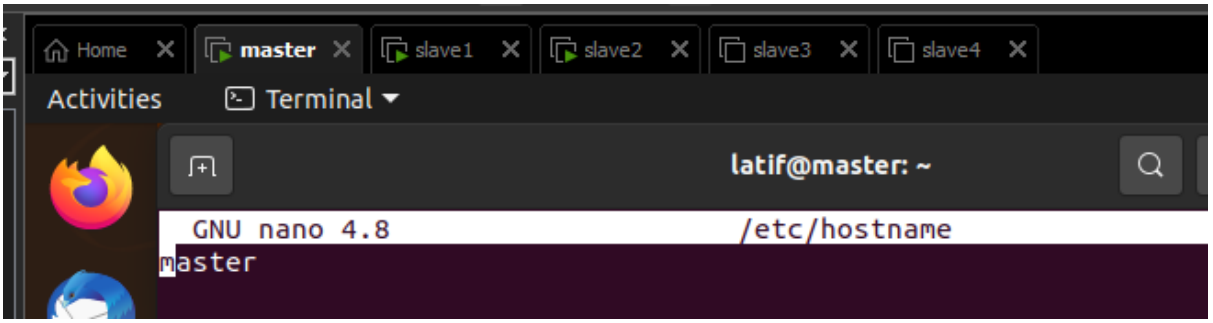



## 2-5 Getting IP addresses and renaming hosts

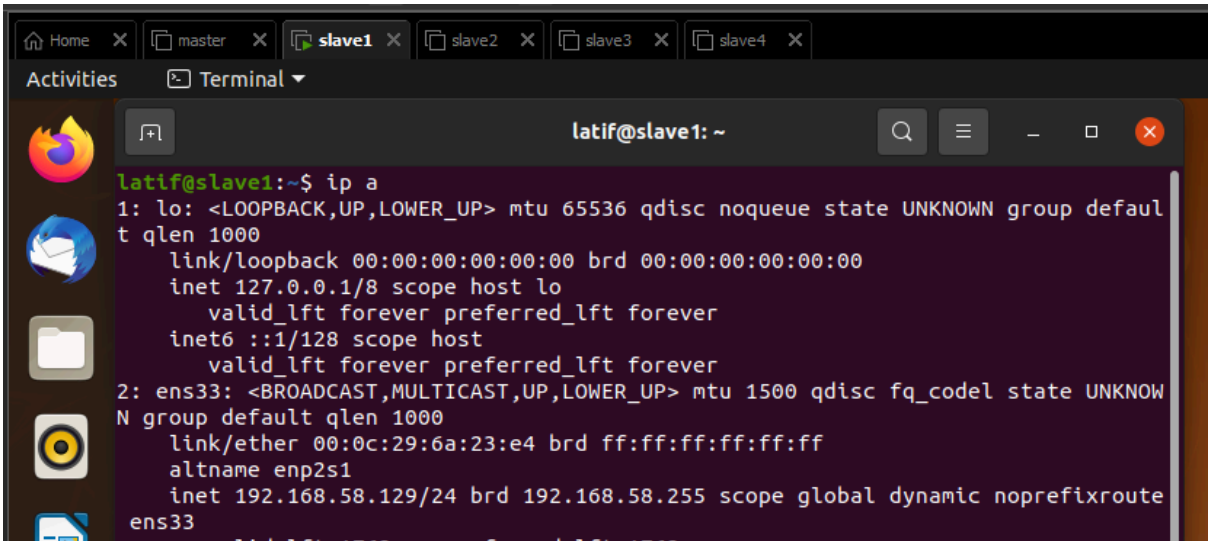
To properly set up the Hadoop cluster, obtaining the IP addresses of all nodes and renaming their hostnames is essential. This step ensures that nodes in the cluster can communicate effectively using hostnames instead of raw IP addresses. Here's how to do it:

- **Edit the Hostname File** Open the /etc/hostname file:
  sudo nano /etc/hostname

Replace the existing hostname with the desired new hostname (e.g., master).



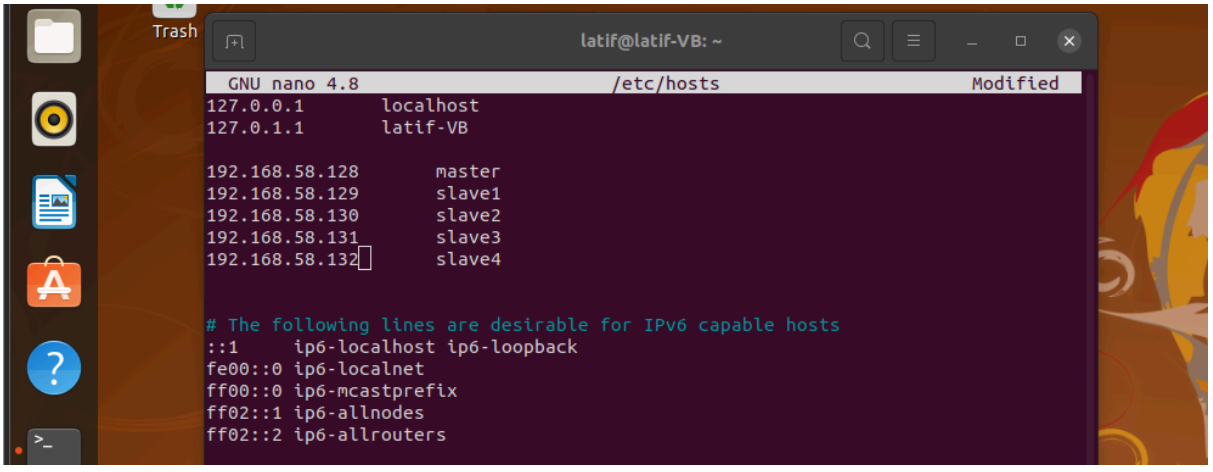

- **Get IP addresses** : Use the ip a ( or ip ad / ip add )




2-6 setting up the hosts files

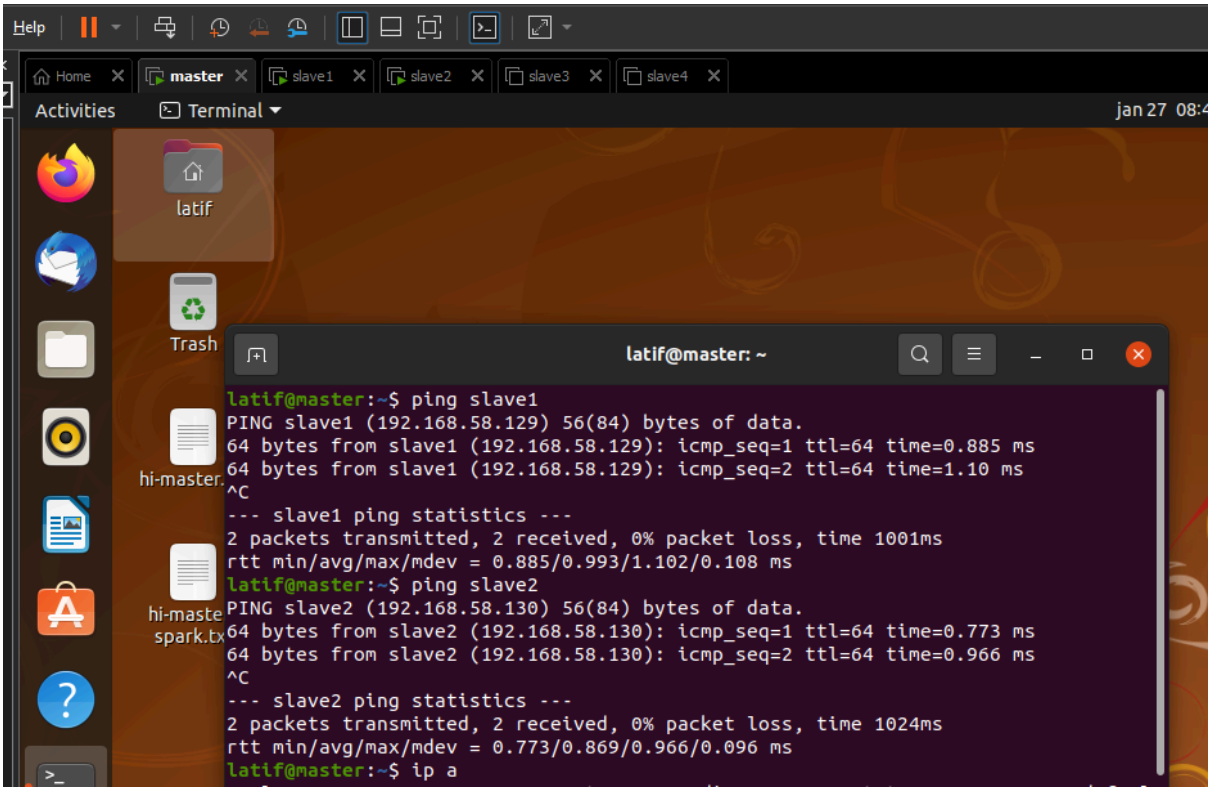
To ensure that each node in the cluster is familiar with the other nodes, we edited the /etc/hosts file on all nodes (master and workers). This step is essential for name resolution within the Hadoop cluster. The following actions were performed:

- **Locate and Edit the Hosts File**
  Open the /etc/hosts file on each node
  *sudo nano /etc/hosts*
- **Add Hostnames and IP Addresses**
  We added entries for all the nodes in the cluster (master and workers), mapping their IP addresses to their hostnames.




- **Verify the Configuration**

We tested the setup by pinging each node from the others using their hostnames

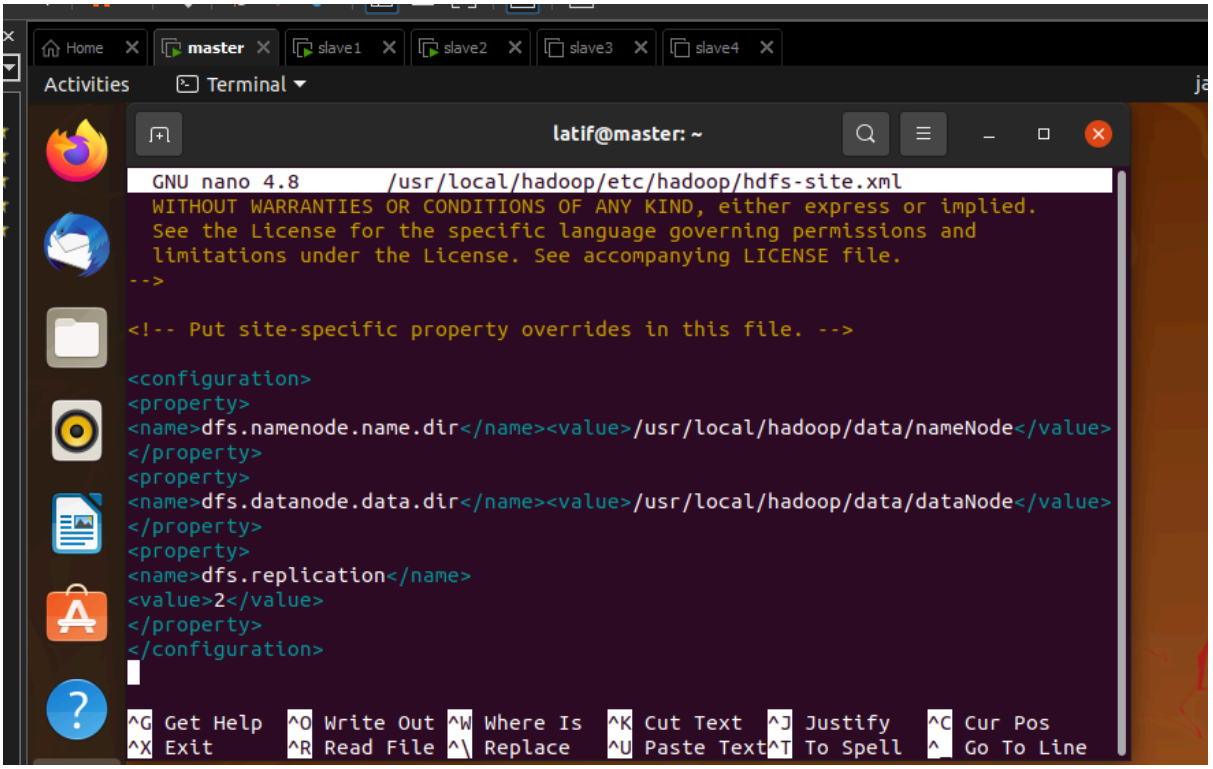



2-7 configuring hadoop on master and streaming the configuration to the slaves

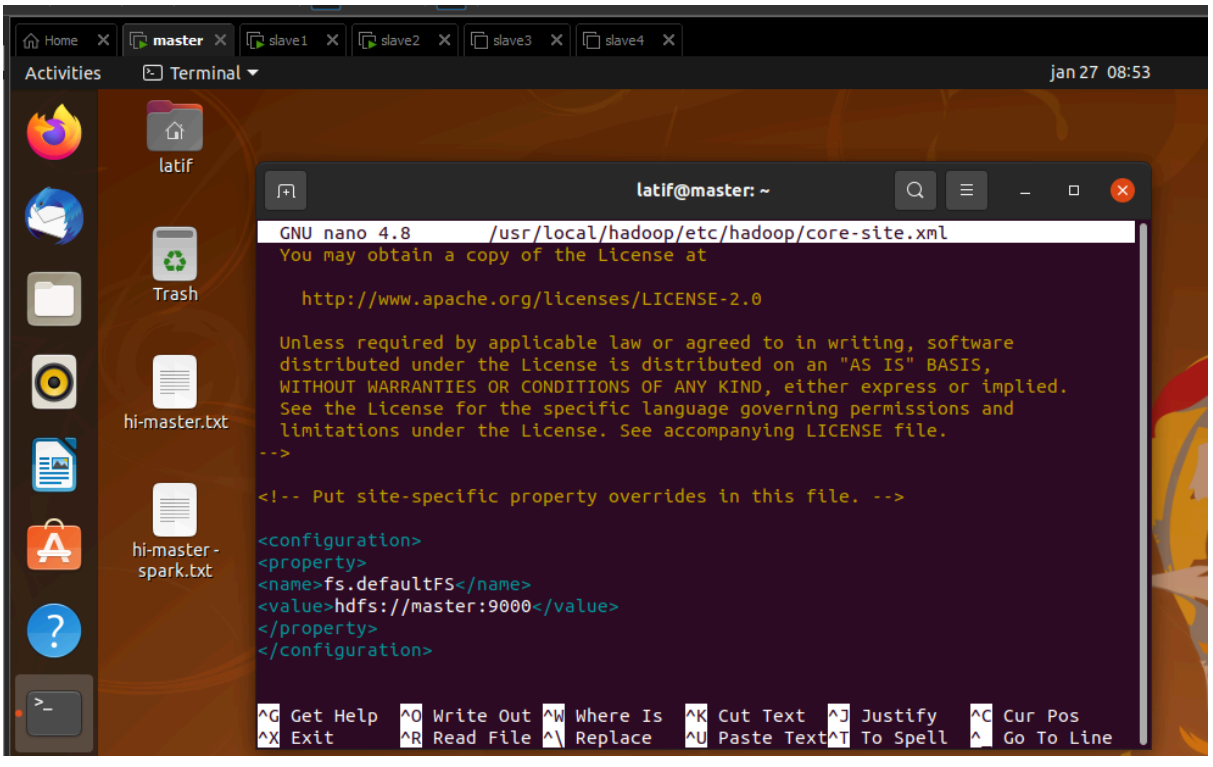
- **Setting up the configuration files and identifying workers**

Inorder to set up a functioning hadoop we need to add the configuration to the hdfs-site.xml , core-site.xml , yarn-site.xml , workers

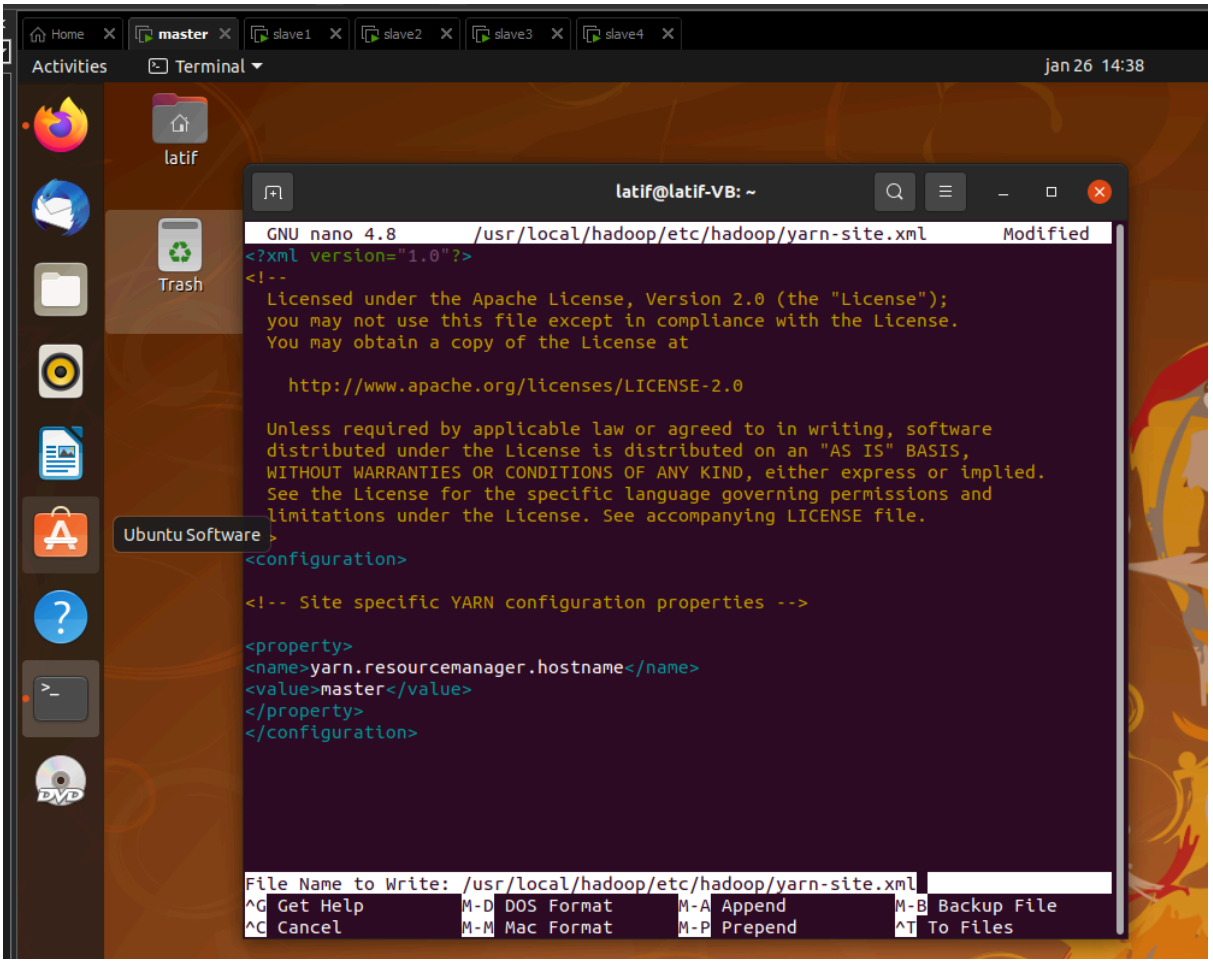
*sudo nano /usr/local/hadoop/etc/hadoop/hdfs-site.xml*



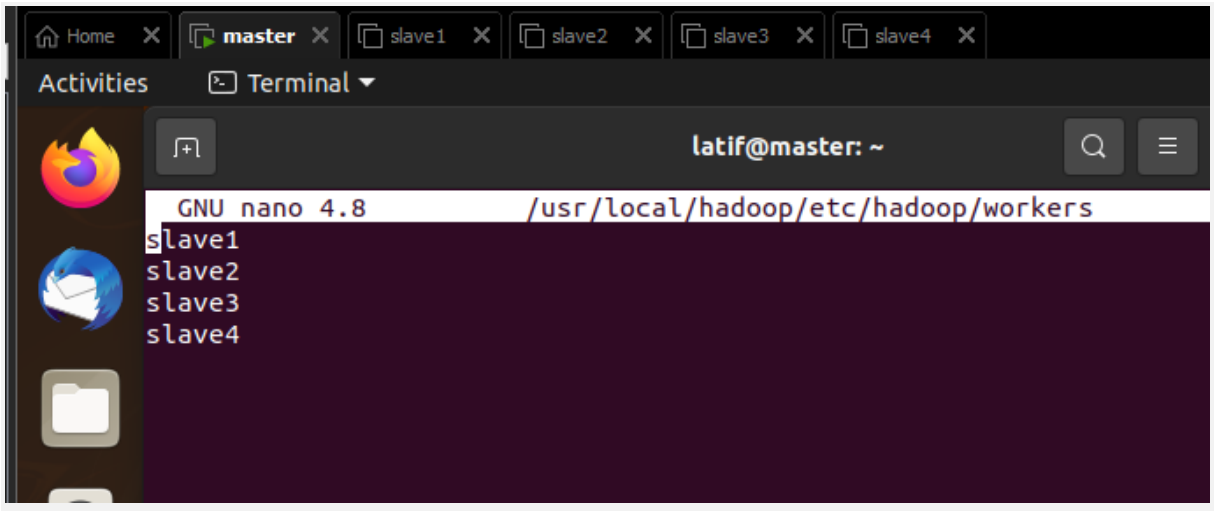

*sudo nano /usr/local/hadoop/etc/hadoop/core-site.xml*




*sudo nano /usr/local/hadoop/etc/hadoop/yarn-site.xml*

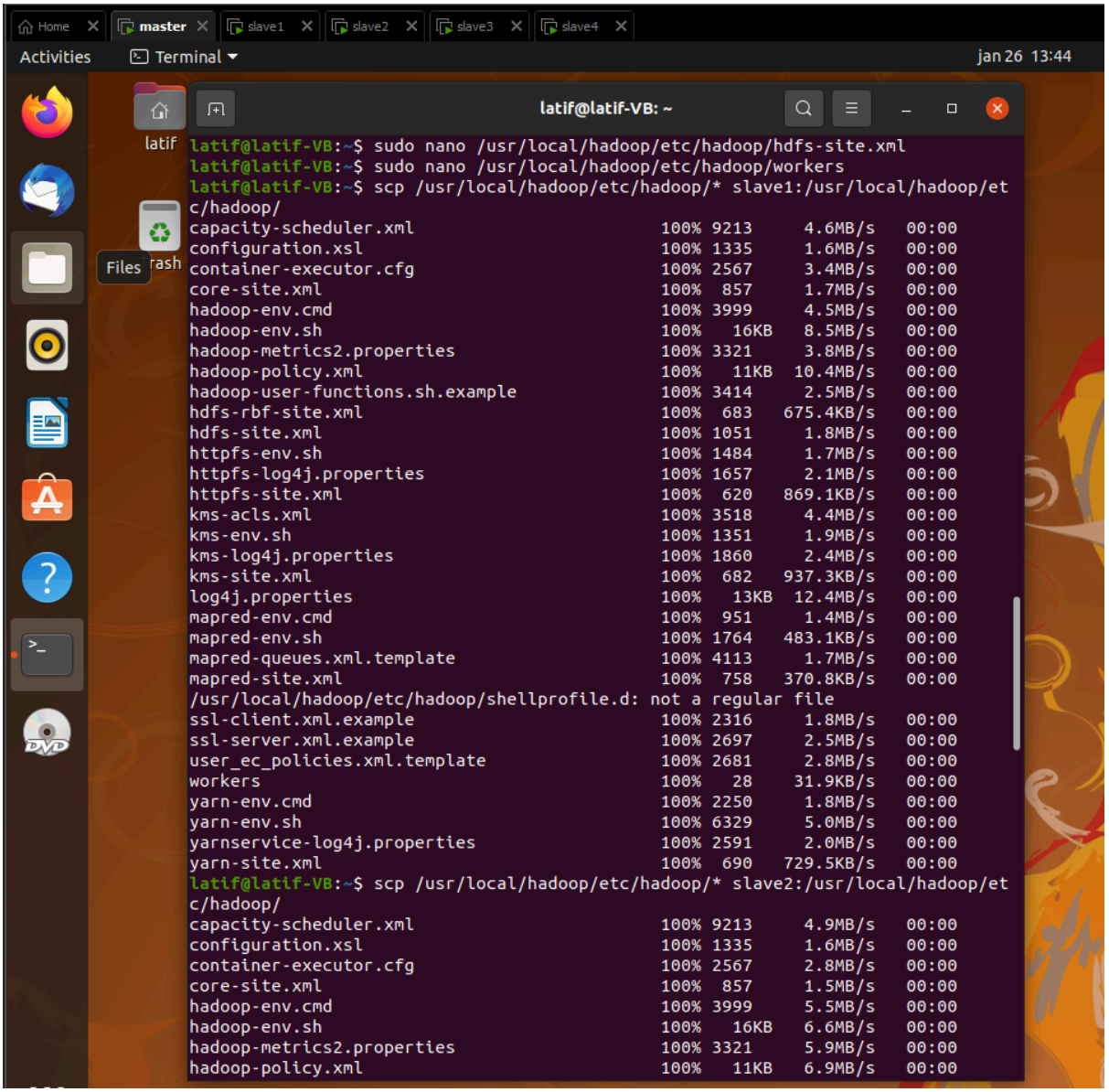



*sudo nano /usr/local/hadoop/etc/hadoop/workers*




- Streaming the updated configuration :
  Using the command to update the new configuration seamlessly to all the other nodes (aka. Slaves ) , to ensure a correct configuration

scp/usr/local/hadoop/etc/hadoop/*h-secondary1:/usr/local/hadoop/etc/hadoop/




2-8 launching hadoop

## Start Hadoop Services

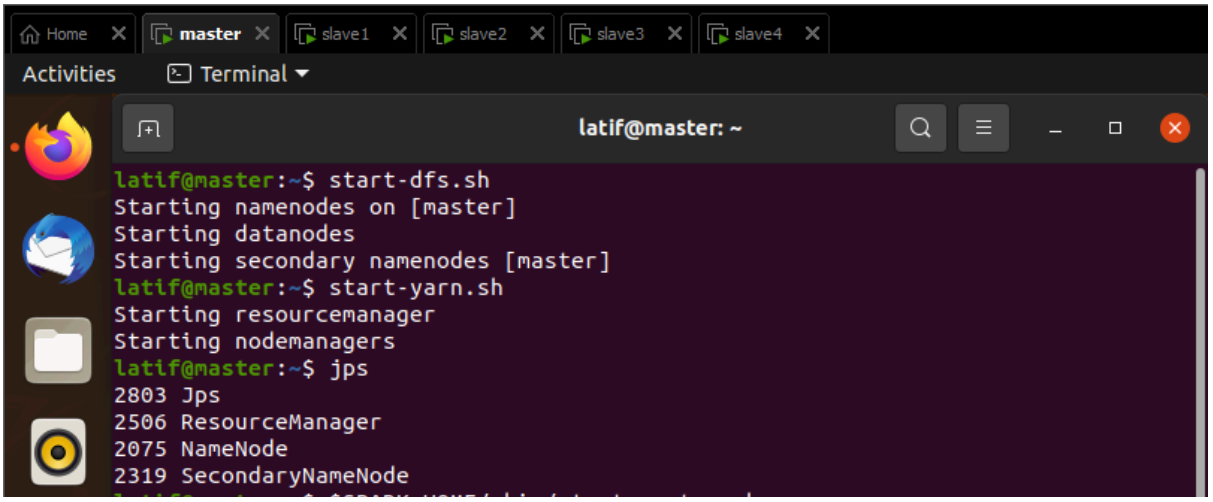
Ensure the Hadoop services are running by executing the following commands on the master node:

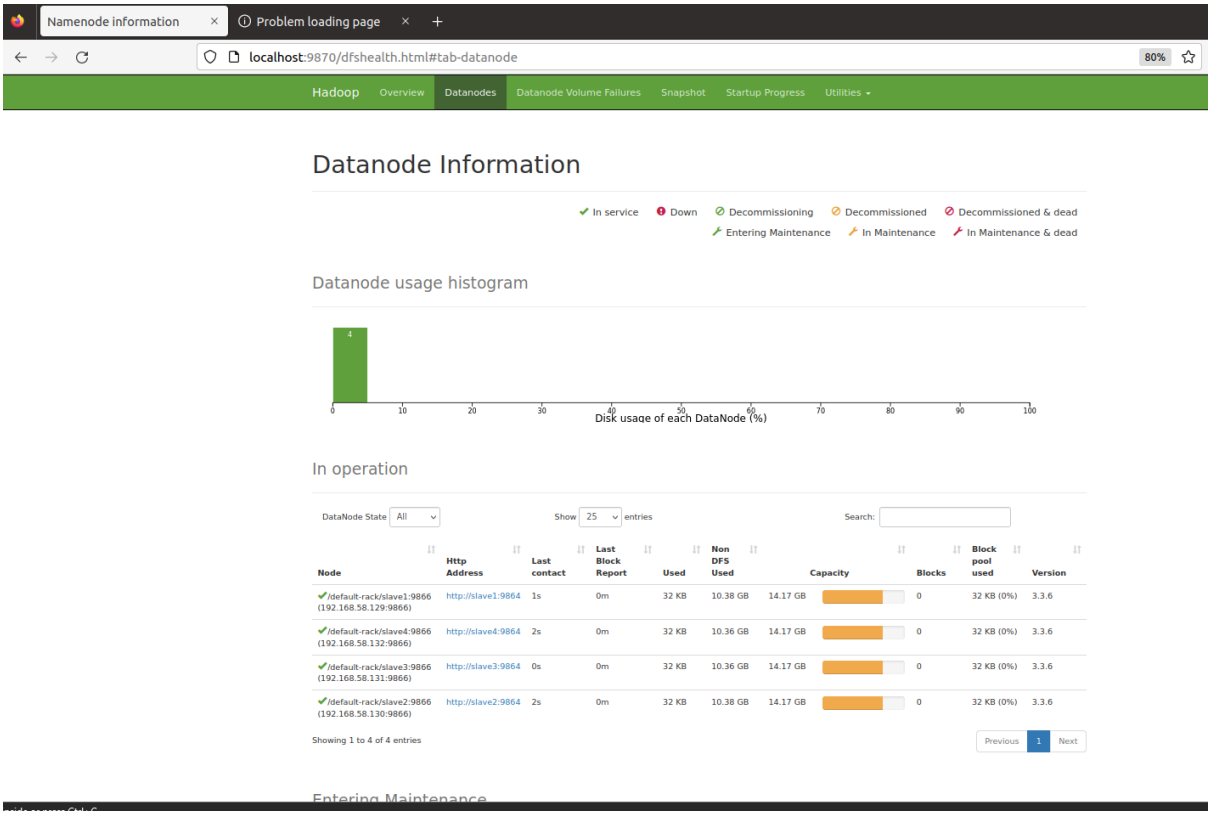
- **Start the Hadoop Distributed File System (HDFS):**

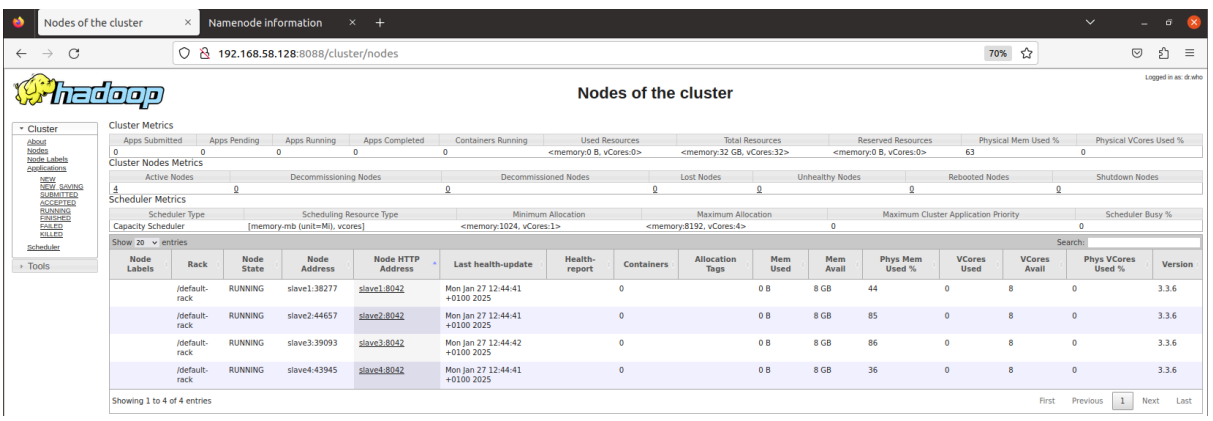
  `start-dfs.sh`
- **Start the YARN resource manager:**

  `start-yarn.sh`
- **Confirm that the services are running** by checking the logs or using:

  `jps`

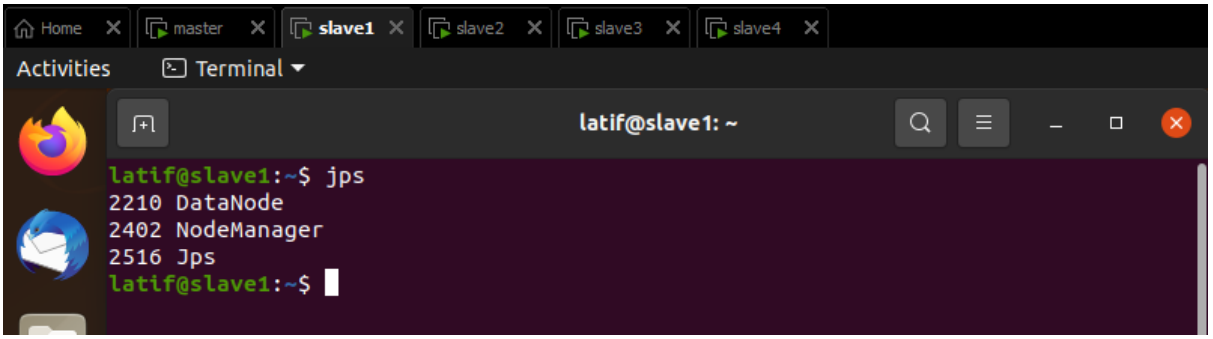



- **Checking hadoop and yarn on slaves**

Using `jps`




## 3- Spark installation and configuration

### 3-1 Installing JAVA

Spark requires Java to operate, and we already have the JDK installed.

### 3-2 Download Apache Spark

Spark requires Java to operate, and we already have the JDK installed.

1. Head over to the Apache Spark Download page.
2. Choose a Spark release:3.4.4 (Oct 27 2024)
3. Choose a package type: Pre-built for Apache Hadoop 3.3 and later Pre-built for Apache Hadoop 3.3 and later (Scala 2.13)
4. Download Spark: spark-3.4.4-bin-hadoop 3.tgz

### 3-3 Installing Spark

- **Extract the Downloaded File**

After downloading the `.tgz` file, the next step is to extract it using the `tar` command

```
tar -xvzf Downloads/spark-3.4.4-bin-hadoop3.tgz
```

- **Rename the spark folder**

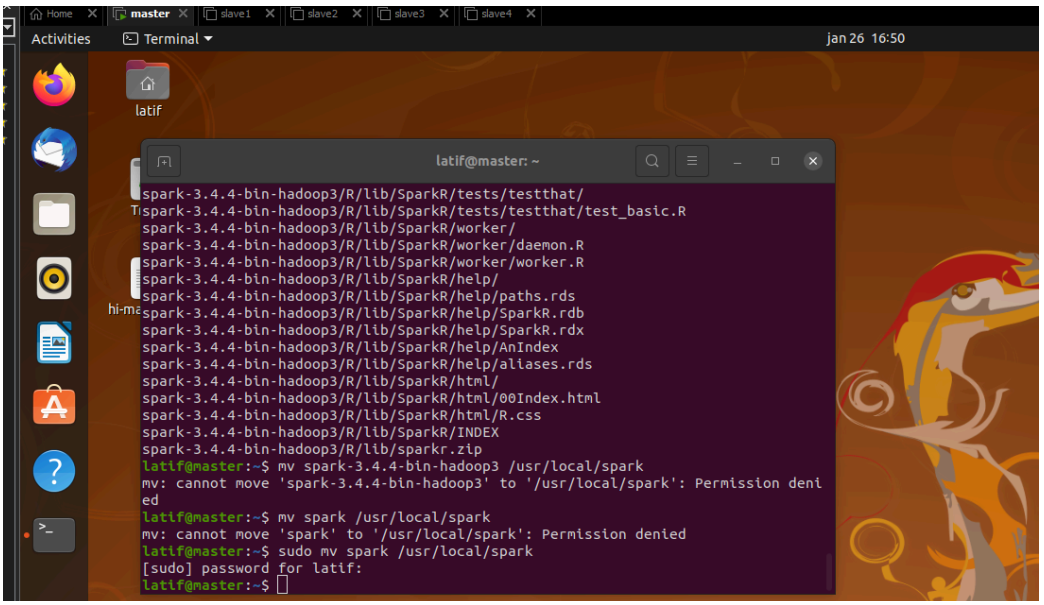
To make it easier for further usage

```
sudo mv spark-3.4.4-bin-hadoop3 spark
```

- **Move the Extracted Folder**

To make Spark accessible system-wide, move the extracted folder to the directory /usr/local.

```
sudo mv spark /usr/local/spark
```




### 3-4 Configure Environment Variables

To ensure that Spark commands are recognized system-wide, update the `.bashrc` file to include Spark's `bin` directory in the system's PATH.

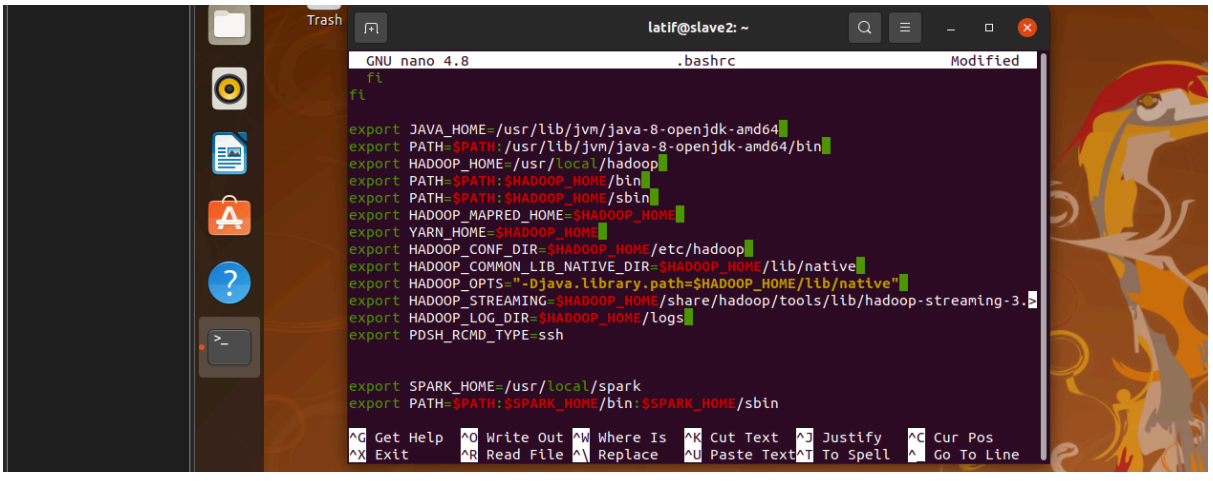
1. Open the `.bashrc` file for editing:

   *sudo nano ~/.bashrc*

2. Add the following lines

   *export SPARK_HOME=/usr/local/spark*

   *export PATH=$SPARK_HOME/bin:$PATH*




### 3-5 Update Workers file

When setting up a Spark cluster, we need to specify the slave nodes (or worker nodes) in the slaves file.

- Open the workers file

  *nano $SPARK_HOME/conf/workers*
- Add the hostnames


3-6 update workers with the new config

- **spark-env.sh File**: Open the file for editing:

  *sudo nano spark-env.sh*
- **Set the Required Environment Variables**: Add the following lines to configure Spark:.

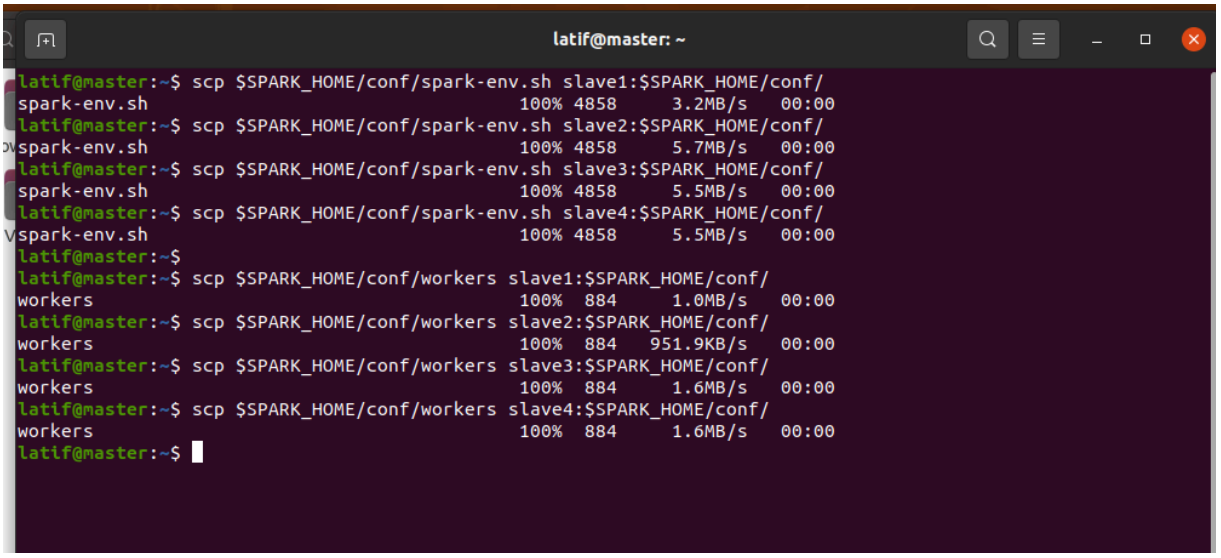
Example configuration:

export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export SPARK_MASTER_HOST=192.168.58.128
export SPARK_WORKER_CORES=4
export SPARK_WORKER_MEMORY=8G
export SPARK_WORKER_INSTANCES=1
export SPARK_LOCAL_DIRS=/usr/local/spark/tmp

**Distribute the spark-env.sh File to All Nodes**: Copy the updated spark-env.sh file to the conf directory on all nodes (workers) in the cluster.

example:

*scp $SPARK_HOME/conf/spark-env.sh slave2:/usr/local/spark/conf/*




## 3-7 launching Spark

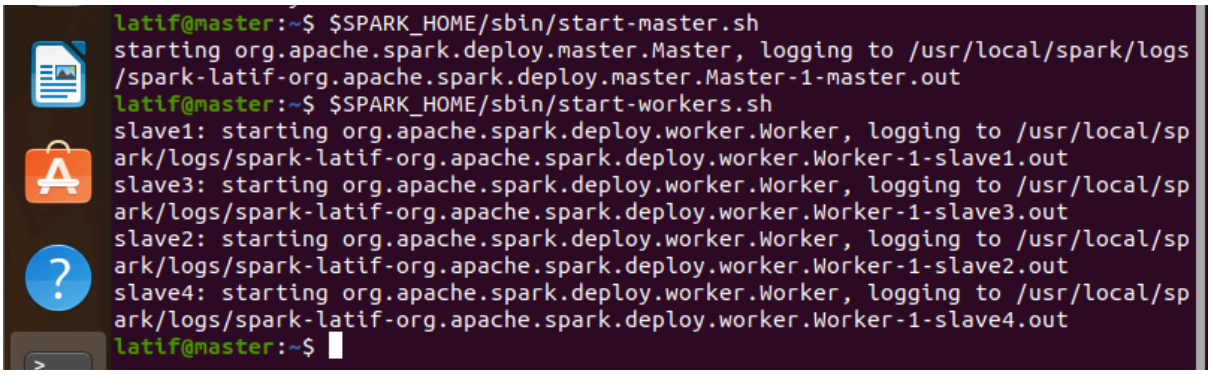
From the master node we run both commands

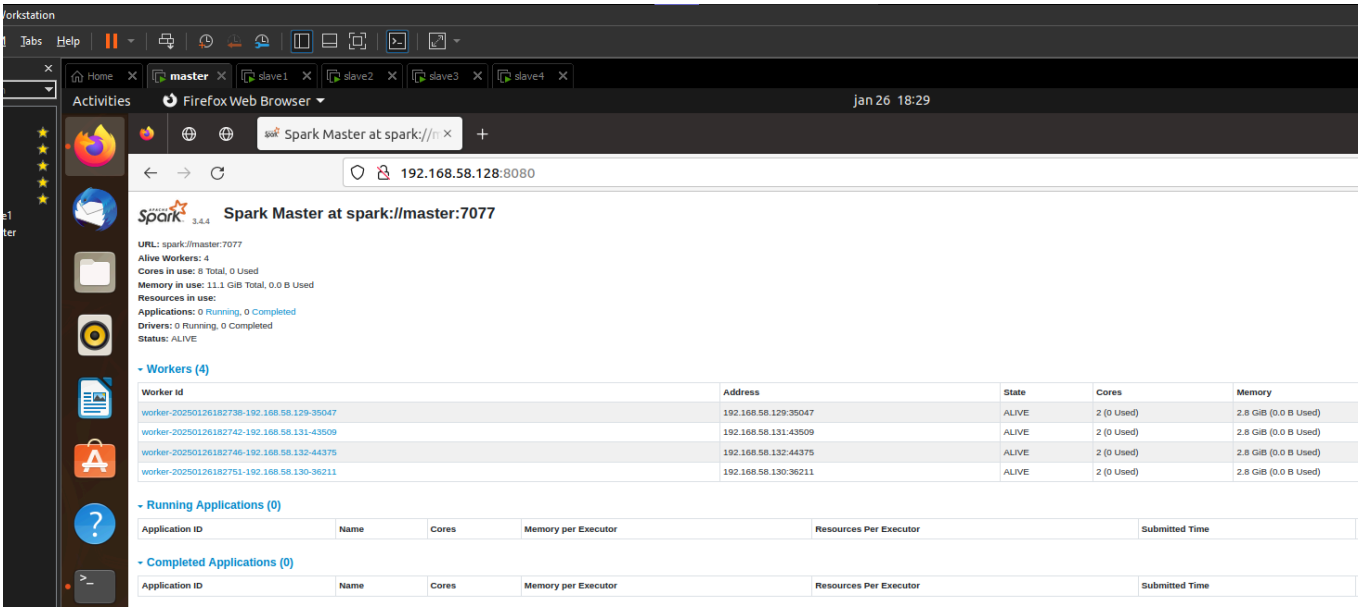
- **To start the master node**

  *$SPARK_HOME/sbin/start-master.sh*

- **To start slaves (workers) nodes**

  *$SPARK_HOME/sbin/start-workers.sh*






# Code and running

## I. First approach : Hadoop & spark

### Code

#### A- Installing and importing the necessary libraries

- Install Python Dependencies

```
sudo apt install -y python3 python3-pip
```

- Install PySpark and dependencies:

```
pip3 install pyspark pillow
```

- Enable JPEG2000 Support in PIL (Pillow)

By default, Pillow **does not support JPEG2000**. You need to install **OpenJPEG**:

```
sudo apt install -y libopenjp2-7-dev
```

#### B- Implementing the solution

1. Initialize SparkSession for connecting to Hadoop cluster
   - Set application name ("JPEG2000 Compression")
   - Set master to "yarn" for cluster management
   - Create Spark context (sc)

2. Define base path on HDFS for storing compressed images
*hdfs_output_base = "hdfs://master:9000/user/hadoop/compressed_images"*

3. Get Hadoop FileSystem handle from Spark for interacting with HDFS
*conf = sc._jsc.hadoopConfiguration()*
*fs = sc._jvm.org.apache.hadoop.fs.FileSystem.get(conf)*
*Path = spark._jvm.org.apache.hadoop.fs.Path  # Java Path class*

4. Define a function `save_to_hdfs(filename, data)`:
- Create full path for the image on HDFS
- Open an output stream to the target HDFS location
- Write the binary data (compressed image) to HDFS
- Close the stream

5. Define a function `compress_image(image_bytes)`:
- Try to open image from binary data
*img = Image.open(io.BytesIO(image_bytes))*
- Compress the image using JPEG2000 format
*output_buffer = io.BytesIO()*
*img.save(output_buffer, format="JPEG2000")*
- Return compressed image as bytes
*return output_buffer.getvalue()*
- Handle errors if any

6. Define the input path for images stored in HDFS
*input_path = "hdfs://master:9000/user/hadoop/input_images"*

7. Load images from HDFS as binary files using Spark's `binaryFiles` function
*images = sc.binaryFiles(input_path)*

8. Map over the loaded images:
- Compress each image using `compress_image`
*compressed_images = images.mapValues(lambda data: compress_image(data))*
- Store compressed data paired with the image filename

9. Collect the results back to the driver (assuming dataset is small enough)
*results = compressed_images.collect()*

10. Loop through the collected results:
- For each image, extract the filename
- Save compressed data to HDFS using `save_to_hdfs`

11. Stop the SparkSession to release resources
*spark.stop()*

## Starting hadoop and spark

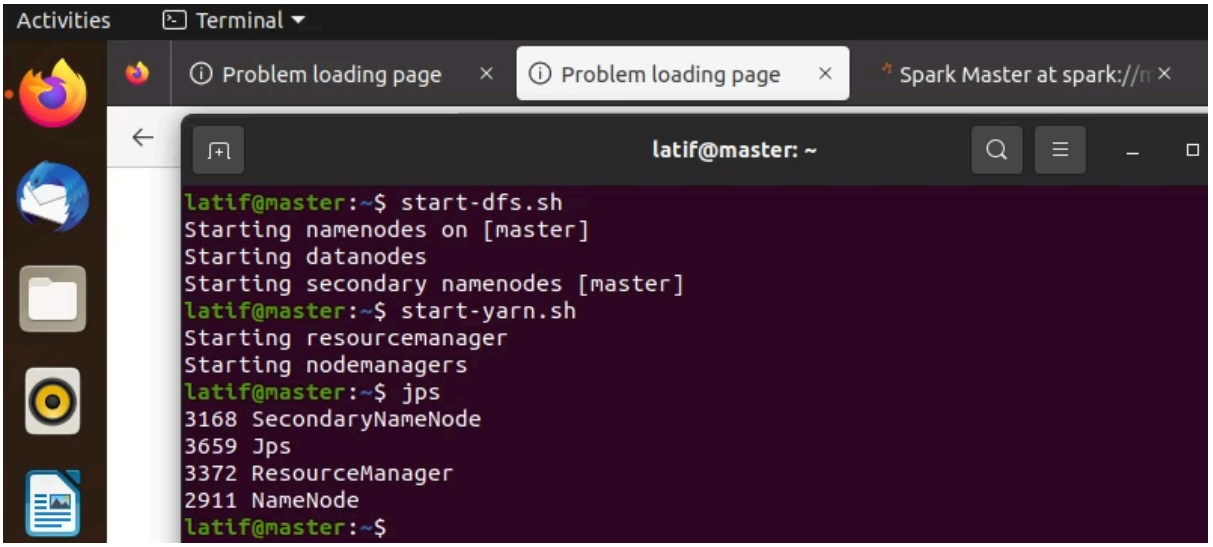
From the master node we start the HDFS and the YARN;

- **Start the Hadoop Distributed File System (HDFS):**

  `start-dfs.sh`

- **Start the YARN resource manager:**

  `start-yarn.sh`




From the master node we run both commands

- **To start the master node**

  *$SPARK_HOME/sbin/start-master.sh*

- **To start slaves (workers) nodes**

  *$SPARK_HOME/sbin/start-workers.sh*


## Check the HDFS

Checking if the *input_images* and *compresed_images* folders exist in the HDFS , if not to be created using

*hdfs dfs -mkdir -p /user/hadoop/compressed_images*
*hdfs dfs -mkdir -p /user/hadoop/input_images*

```
latif@master:~$ hdfs dfs -ls /user/hadoop/
Found 3 items
drwxr-xr-x   - latif supergroup          0 2025-02-01 21:11 /user/hadoop/compressed_images
drwxr-xr-x   - latif supergroup          0 2025-02-01 20:41 /user/hadoop/input_images
drwxr-xr-x   - latif supergroup          0 2025-02-01 17:34 /user/hadoop/jars
latif@master:~$
```

## Upload the images to the designated folder in HDFS

Using the command below allows the upload of the files from the local machine to the HDFS system

*hdfs dfs -put ~/Desktop/Hadoop-job/images/* /user/hadoop/input_images*

```
latif@master:~/Desktop/mapreduce-job$ hdfs dfs -put ~/Desktop/Hadoop-job/images/* /user/hadoop/input_images
latif@master:~/Desktop/mapreduce-job$ hdfs dfs -ls /user/hadoop/input_images
Found 3 items
-rw-r--r--   2 latif supergroup    2108235 2025-02-01 23:18 /user/hadoop/input_images/20241231_111322.jpg
-rw-r--r--   2 latif supergroup    2464886 2025-02-01 23:18 /user/hadoop/input_images/20250107_093226.jpg
-rw-r--r--   2 latif supergroup     240045 2025-02-01 23:18 /user/hadoop/input_images/IMG_20241231_202142_665.jpg
```

## Job submission

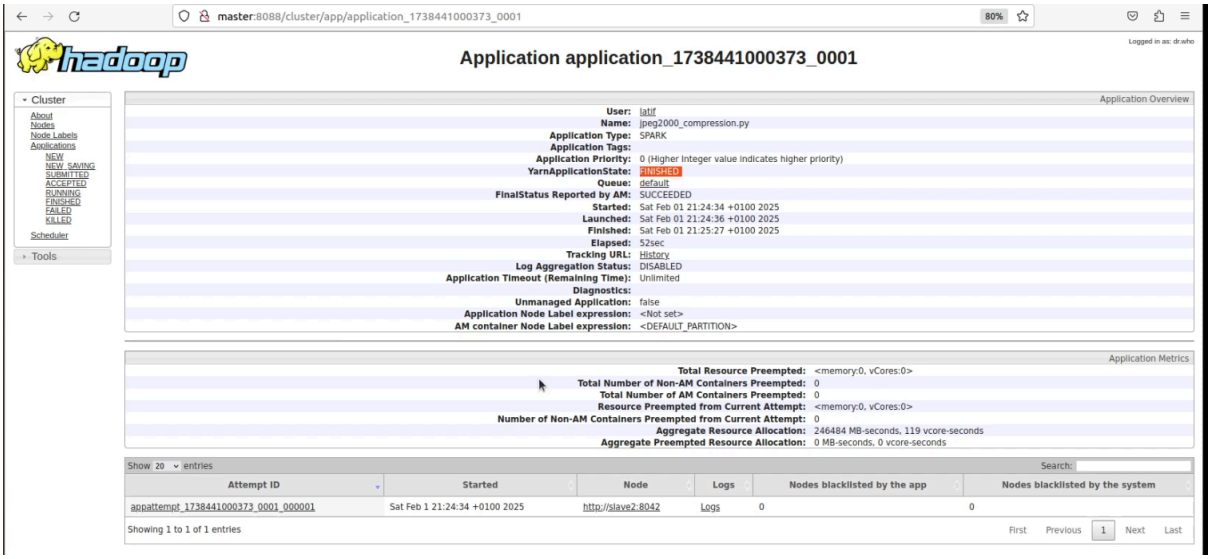
`spark-submit` command, it communicates with the cluster manager ( YARN) to allocate resources on available nodes.

*spark-submit --master yarn --deploy-mode cluster Desktop/Hadoop-job/jpeg2000_compression.py*

```
latif@master:~$ spark-submit --master yarn --deploy-mode cluster Desktop/Hadoop-job/jpeg2000_compression.py
```

## Running and waiting for job compilation

When a Spark job is submitted, it is compiled and prepared for execution across the cluster's nodes. Once the compilation is complete, the job is scheduled by the cluster manager, and tasks are distributed to available worker nodes for parallel processing.




## Retrieving results

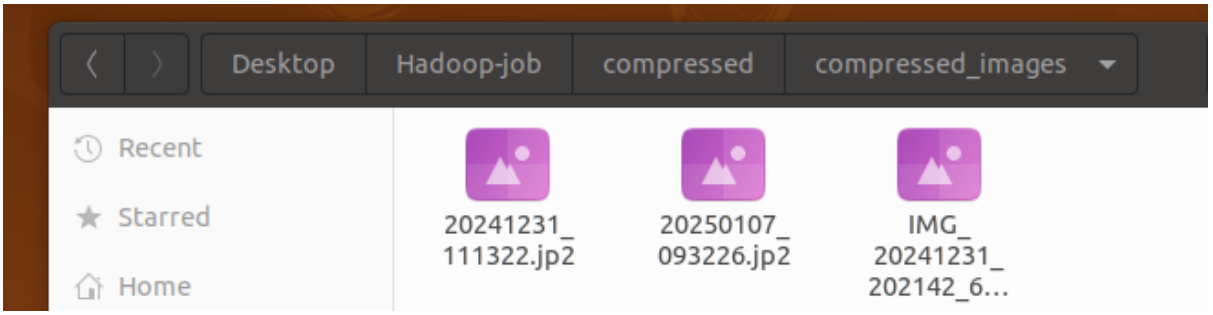
We can check if the results of the job (in our case the compression job ) gave a result or no by checking the designated folder for saving the results on the HDFS ,by using the command

*hdfs dfs -ls /user/hadoop/compressed_images*

```
latif@master:~$ hdfs dfs -ls /user/hadoop/compressed_images
Found 3 items
-rw-r--r--   2 latif supergroup   10040834 2025-02-01 21:11 /user/hadoop/compres
sed_images/20241231_111322.jp2
-rw-r--r--   2 latif supergroup   11141779 2025-02-01 21:11 /user/hadoop/compres
sed_images/20250107_093226.jp2
-rw-r--r--   2 latif supergroup    2188856 2025-02-01 21:11 /user/hadoop/compres
sed_images/IMG_20241231_202142_665.jp2
```

The results can be saved on one of the local machines just by running

*hdfs dfs -get /user/hadoop/compressed_images ~/Desktop/Hadoop-job/compressed*



# II. Second approach : Hadoop (Map-Reduce)

## Code

Three parts of code were implemented supporting **JPEG** compression to assure the functionality of the map reduce job

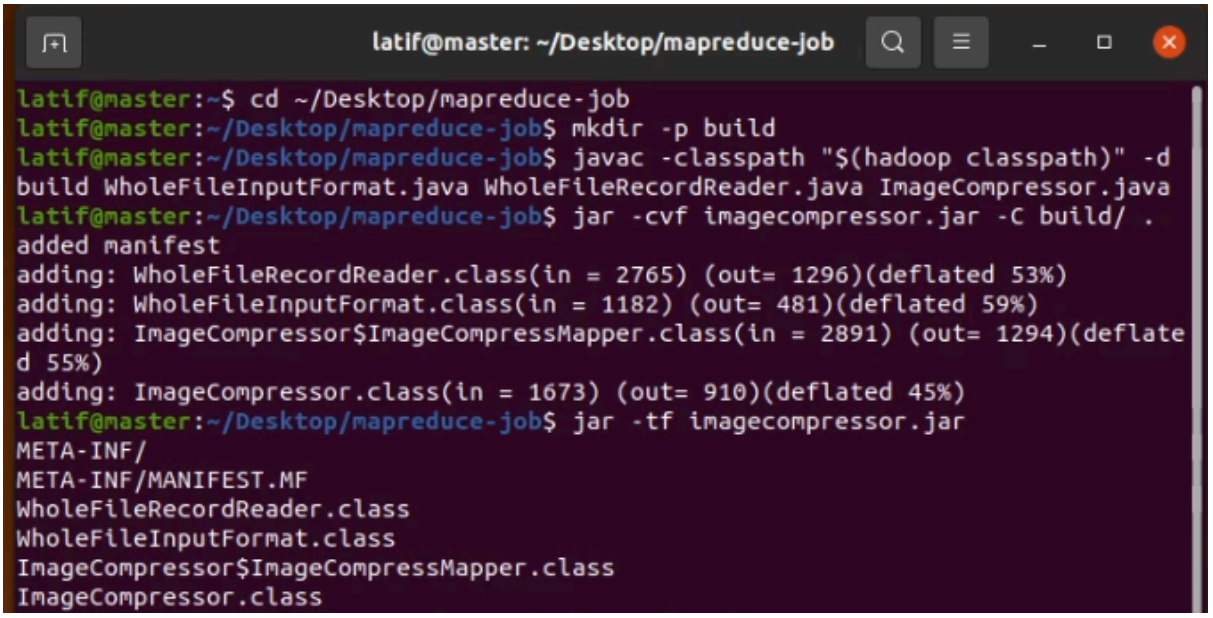

| code | WholeFileInputFormat.java | WholeFileRecordReader.java | ImageCompressor.java |
|---|---|---|---|
| job | This custom input format ensures that each file is treated as a single input record instead of being split into smaller chunks | This custom record reader reads an entire file into memory and stores its contents as a BytesWritable object. | This file contains:<br>1. ImageCompressMapper (Mapper class) → Responsible for compressing images.<br>2. Main method → Configures and runs the MapReduce job. |
| Key points | ● Extends FileInputFormat<Text, BytesWritable> to specify the input format.<br>● Overrides isSplitable() → returns false, meaning that each file is processed as a whole.<br>● Overrides createRecordReader() → provides a WholeFileRecordReader to read the file as a whole. | ● Reads the entire file into a byte array.<br>● Stores the file path as the key (Text) and file content as the value (BytesWritable).<br>● Ensures that each file is processed only once (processed flag). | ImageCompressMapper<br>● Reads the image from BytesWritable.<br>● Uses ImageIO to convert it to a BufferedImage.<br>● Compresses the image to JPEG format with 50% quality.<br>● Writes the compressed image data as output.<br>Main Method<br>● Sets up a Hadoop MapReduce job.<br>● Uses WholeFileInputFormat to read full image files.<br>● Runs only the Mapper function (map-only job) with setNumReduceTasks(0).<br>● Takes input and output paths as arguments |
| Importance | ● Ensures that image files are processed completely rather than split, which is crucial for image processing. | ● Essential for handling binary files like images, ensuring the entire file is available for processing. | ● Image compression reduces storage size while maintaining reasonable quality.<br>● Efficient processing with a map-only job, ideal for large-scale distributed image compression |

## Compiling the java files

To prepare the previous files to be used in the job , the task of compiling them is assured by javac

*javac -classpath $(hadoop classpath) -d build/ WholeFileInputFormat.java WholeFileRecordReader.java ImageCompressor.java*

- After compilation , we need to build the jar that's going to be introduced later on




## Starting hadoop

From the master node we start the HDFS and the YARN;

- **Start the Hadoop Distributed File System (HDFS):**

  `start-dfs.sh`

- **Start the YARN resource manager:**

  `start-yarn.sh`

## Check the HDFS

Checking if the *input_images* and *compresed_images* folders exist in the HDFS , if not to be created using

*hdfs dfs -mkdir -p /user/hadoop/compressed_images*
*hdfs dfs -mkdir -p /user/hadoop/input_images*

## Upload the images to the designated folder in HDFS

Using the command below allows the upload of the files from the local machine to the HDFS system

*hdfs dfs -put ~/Desktop/Hadoop-job/images/* /user/hadoop/input_images*

```
latif@master:~/Desktop/mapreduce-job$ hdfs dfs -put ~/Desktop/Hadoop-job/images/* /user/hadoop/input_images
latif@master:~/Desktop/mapreduce-job$ hdfs dfs -ls /user/hadoop/input_images
Found 3 items
-rw-r--r--   2 latif supergroup    2108235 2025-02-01 23:18 /user/hadoop/input_images/20241231_111322.jpg
-rw-r--r--   2 latif supergroup    2464886 2025-02-01 23:18 /user/hadoop/input_images/20250107_093226.jpg
-rw-r--r--   2 latif supergroup     240045 2025-02-01 23:18 /user/hadoop/input_images/IMG_20241231_202142_665.jpg
```

## Job submission

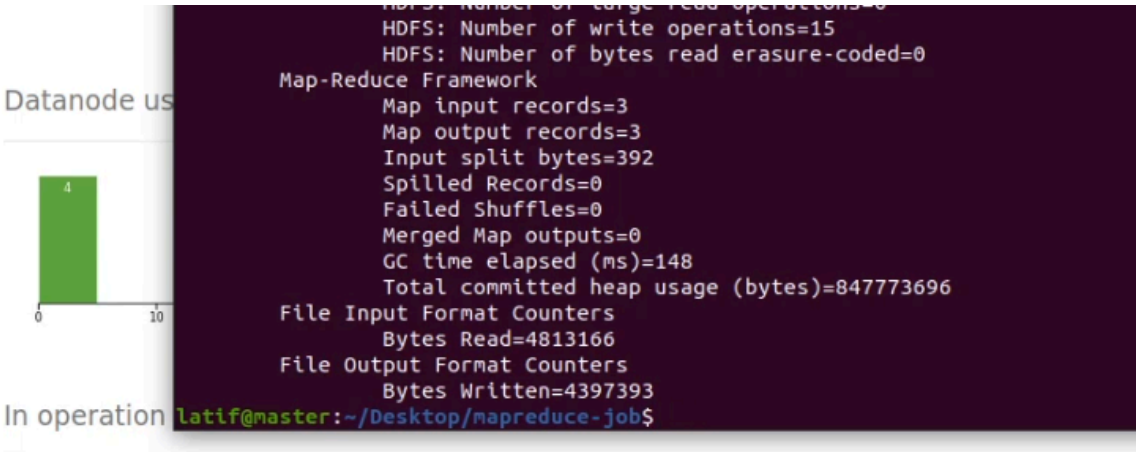
`Hadoop jar` command communicates with the cluster manager ( YARN) to allocate resources on available nodes.

*hadoop jar imagecompressor.jar ImageCompressor /user/hadoop/input_images /user/hadoop/compressed_images*

```
latif@master:~/Desktop/mapreduce-job$ hadoop jar /home/latif/Desktop/mapreduce-job/imagecompressor.jar ImageCompressor /user/hadoop/input_images /user/hadoop/compressed_images
2025-02-01 23:19:28,371 INFO impl.MetricsConfig: Loaded properties from hadoop-metrics2.properties
2025-02-01 23:19:28,508 INFO impl.MetricsSystemImpl: Scheduled Metric snapshot period at 10 second(s).
2025-02-01 23:19:28,508 INFO impl.MetricsSystemImpl: JobTracker metrics system started
```

## Running and waiting for job compilation

When a mapreduce job is submitted, it is compiled and prepared for execution across the cluster's nodes. Once the compilation is complete, the job is scheduled by the cluster manager, and tasks are distributed to available worker nodes for parallel processing.




## Downloading the output files

Using the command

*hdfs dfs -get /user/hadoop/compressed_images ~/Desktop/mapreduce-job/compressed*

```
Bytes Written=4397393
latif@master:~/Desktop/mapreduce-job$ hdfs dfs -get /user/hadoop/compressed_images ~/Desktop/mapreduce-job/compressed
```

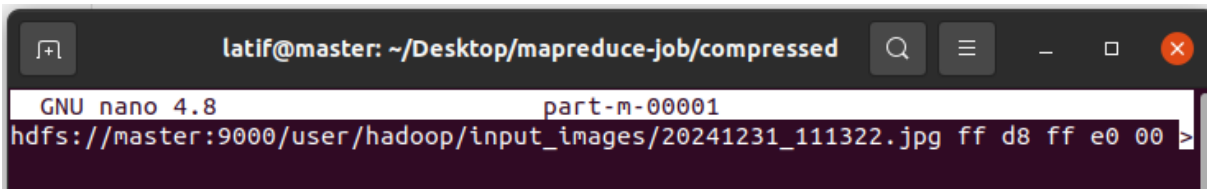
### Fix the output file to see the results

Hadoop outputs images data in an incompatible format (e.g., hex or with extra metadata). This process ensures we get a clean binary JPEG file.

To visualise the file , we need to go through the following steps

-open image using sudo nano

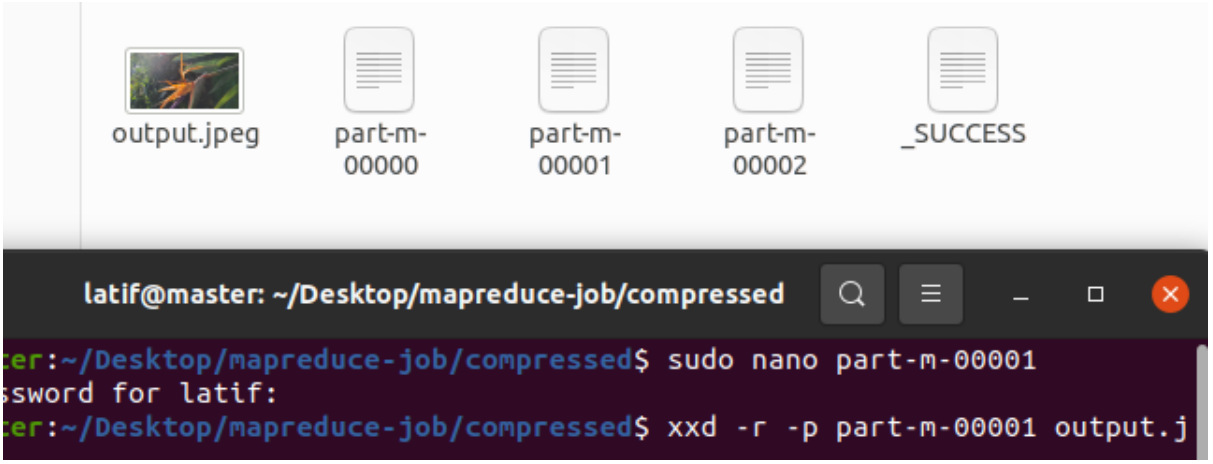
-remove the first line which can look something like




-use this after to reproduce

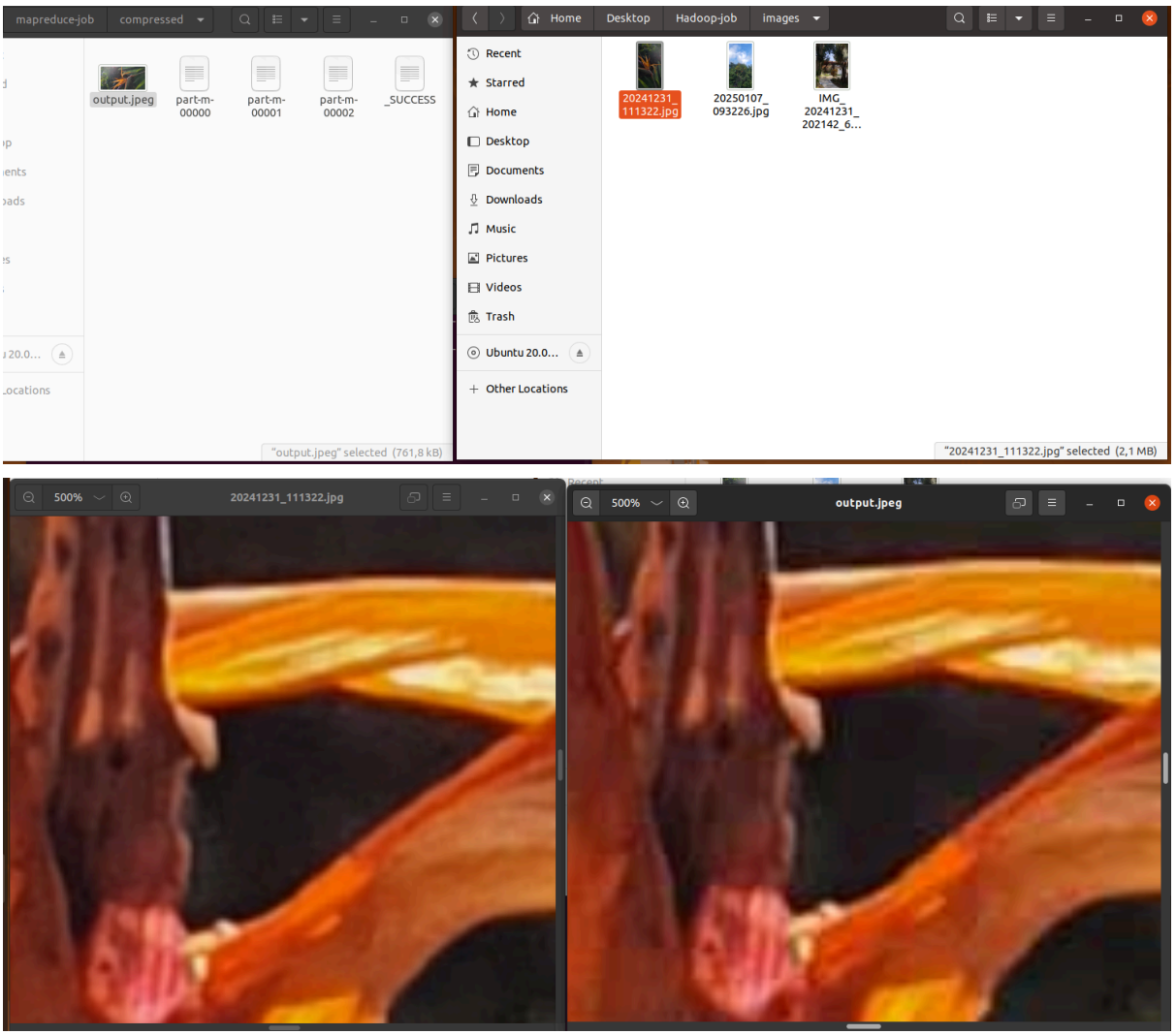
*xxd -r -p part-m-00001 output.jpeg*

–And get something like this after




# Results

The compression effectively reduced image file sizes while maintaining high visual quality. For example, a **flower image originally sized at 2.1MB was compressed to 761.8KB**, demonstrating a significant reduction in storage space. At normal viewing distances, the quality appeared nearly identical to the original. However, upon zooming beyond **400% scale**, subtle differences became noticeable, such as minor loss of fine details and texture. This result confirms that **JPEG achieves high compression efficiency while preserving essential image quality**, making it a viable solution for reducing storage needs without a significant visual downgrade.




# Conclusion and Discussion

In this project, we compared two approaches for processing and compressing images using: JPEG **MapReduce alone** and JPEG2000 on **Hadoop with Spark**. Both methods successfully performed the task, demonstrating the effectiveness of distributed computing for handling large-scale data.

When running MapReduce, the execution was relatively smooth on our setup, and the system remained responsive throughout the process. On the other hand, running Hadoop with Spark introduced significant delays before the actual job execution. The initialization phase of Spark, including job scheduling and resource allocation, caused noticeable slowdowns, even leading to system freezes at times.

However, despite the performance issues observed in our specific setup—running five virtual machines on a single host—both approaches yielded promising results. It is important to consider that Spark is designed for large-scale distributed processing and typically requires **more powerful hardware** to fully leverage its advantages, such as in-memory

computation and optimized parallel execution. If executed on a more capable cluster, **Spark's performance could have been significantly better**, potentially surpassing MapReduce in efficiency and speed.

Ultimately, while our findings suggest that MapReduce was more stable on our limited hardware, we **cannot definitively say one approach is superior** based solely on this experiment. Both technologies are built to handle **big data challenges**, which typically require enterprise-grade infrastructure rather than a modest personal setup. In a more powerful environment, Spark could demonstrate its full potential, making it a strong alternative to traditional MapReduce for large-scale data processing

# Notes

To provide further insight into the project, the following resources are available:

- **Demo Videos:** Demo videos
    - This includes recorded demonstrations showcasing the model in action, its predictions, and key results. It provides a visual overview of how the approach performs in real-world scenarios.
- **GitHub Repository:** Github repository
    - The full implementation, including data preprocessing, model training, evaluation scripts, and additional resources, can be found in this repository. Feel free to explore the code, run experiments, or contribute improvements.