1. Cluster Setup:

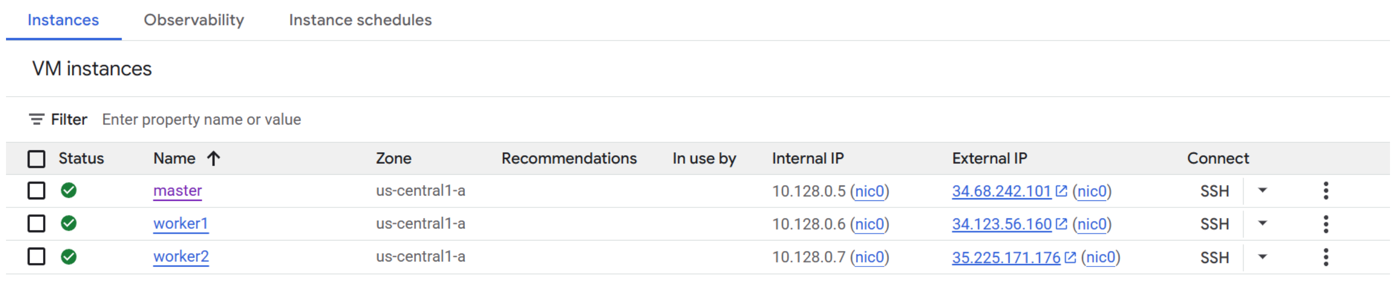
1.1 Creating the Virtual Machines (Master and Workers)

To build the Hadoop and Spark cluster, I used **Google Cloud Platform (GCP) Compute Engine**. I created **three virtual machines**: one master node and two worker nodes. All three nodes were created with **identical specifications** to ensure compatibility across Hadoop and Spark.

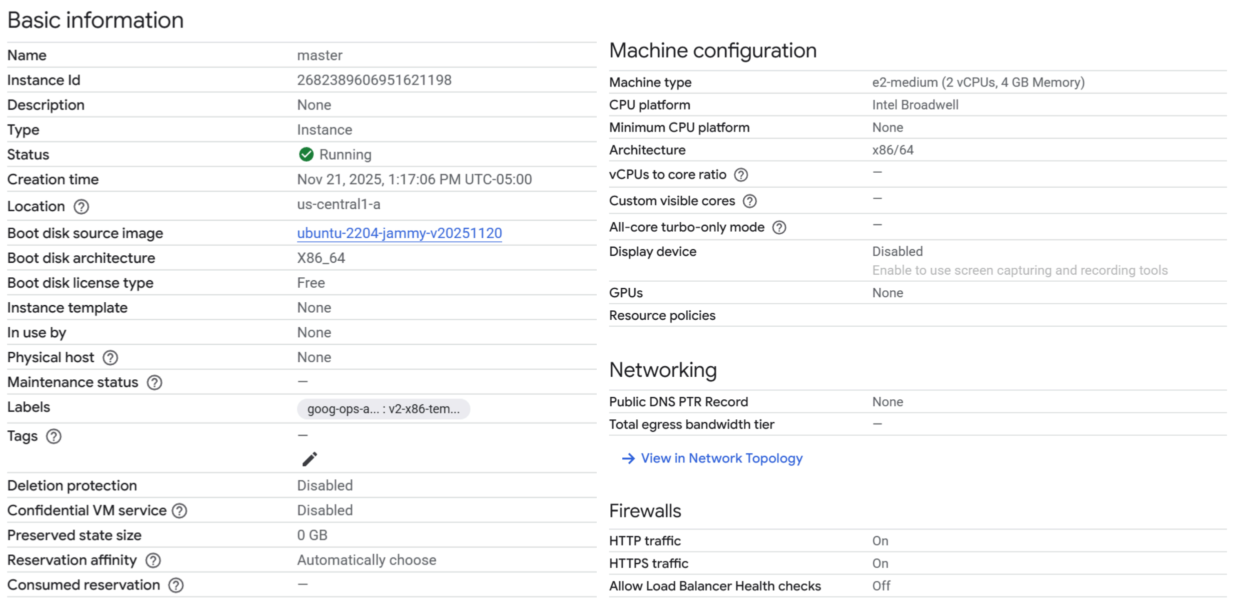
VM Specifications (for all nodes: master, worker1, worker2)

- **Operating System:** Ubuntu 22.04 LTS
- **Architecture:** x86/64
- **Machine Type:** e2-medium (2 vCPUs, 4 GB RAM)
- **CPU Platform:** Intel Broadwell
- **Boot Disk:** 10 GB Balanced Persistent Disk
- **Zone:** us-central1-a
- **Network:** default VPC network, default subnetwork
- **Firewall Rules:** HTTP and HTTPS allowed
- **Public IP:** Ephemeral
- **Internal IP:** Automatically assigned in the 10.128.0.x range

These settings match the requirements for Hadoop and Spark, since both are optimized for **x86-64 architectures** and expect Java-based environments.



Status	Name	Zone	Recommendations	In use by	Internal IP	External IP	Connect
✓	master	us-central1-a			10.128.0.5 (nic0)	34.68.242.101 (nic0)	SSH
✓	worker1	us-central1-a			10.128.0.6 (nic0)	34.123.56.160 (nic0)	SSH
✓	worker2	us-central1-a			10.128.0.7 (nic0)	35.225.171.176 (nic0)	SSH



Basic information

Name	master
Instance Id	2682389606951621198
Description	None
Type	Instance
Status	Running
Creation time	Nov 21, 2025, 1:17:06 PM UTC-05:00
Location	us-central1-a
Boot disk source image	ubuntu-2204-jammy-v20251120
Boot disk architecture	X86_64
Boot disk license type	Free
Instance template	None
In use by	None
Physical host	None
Maintenance status	—
Labels	goog-ops-a... : v2-x86-tem...
Tags	—
Deletion protection	Disabled
Confidential VM service	Disabled
Preserved state size	0 GB
Reservation affinity	Automatically choose
Consumed reservation	—

Machine configuration

Machine type	e2-medium (2 vCPUs, 4 GB Memory)
CPU platform	Intel Broadwell
Minimum CPU platform	None
Architecture	x86/64
vCPUs to core ratio	—
Custom visible cores	—
All-core turbo-only mode	—
Display device	Disabled Enable to use screen capturing and recording tools
GPUs	None
Resource policies	

Networking

Public DNS PTR Record	None
Total egress bandwidth tier	—

Firewalls

HTTP traffic	On
HTTPS traffic	On
Allow Load Balancer Health checks	Off

This matters because all nodes must be identical in CPU architecture, OS, and Java compatibility so that Hadoop and Spark binaries behave consistently across the cluster and do not fail due to environment mismatch.

1.2 Initial System Setup on All Nodes

After creating each VM, I connected via SSH (from the GCP console) and installed the necessary system packages required for Hadoop and Spark.

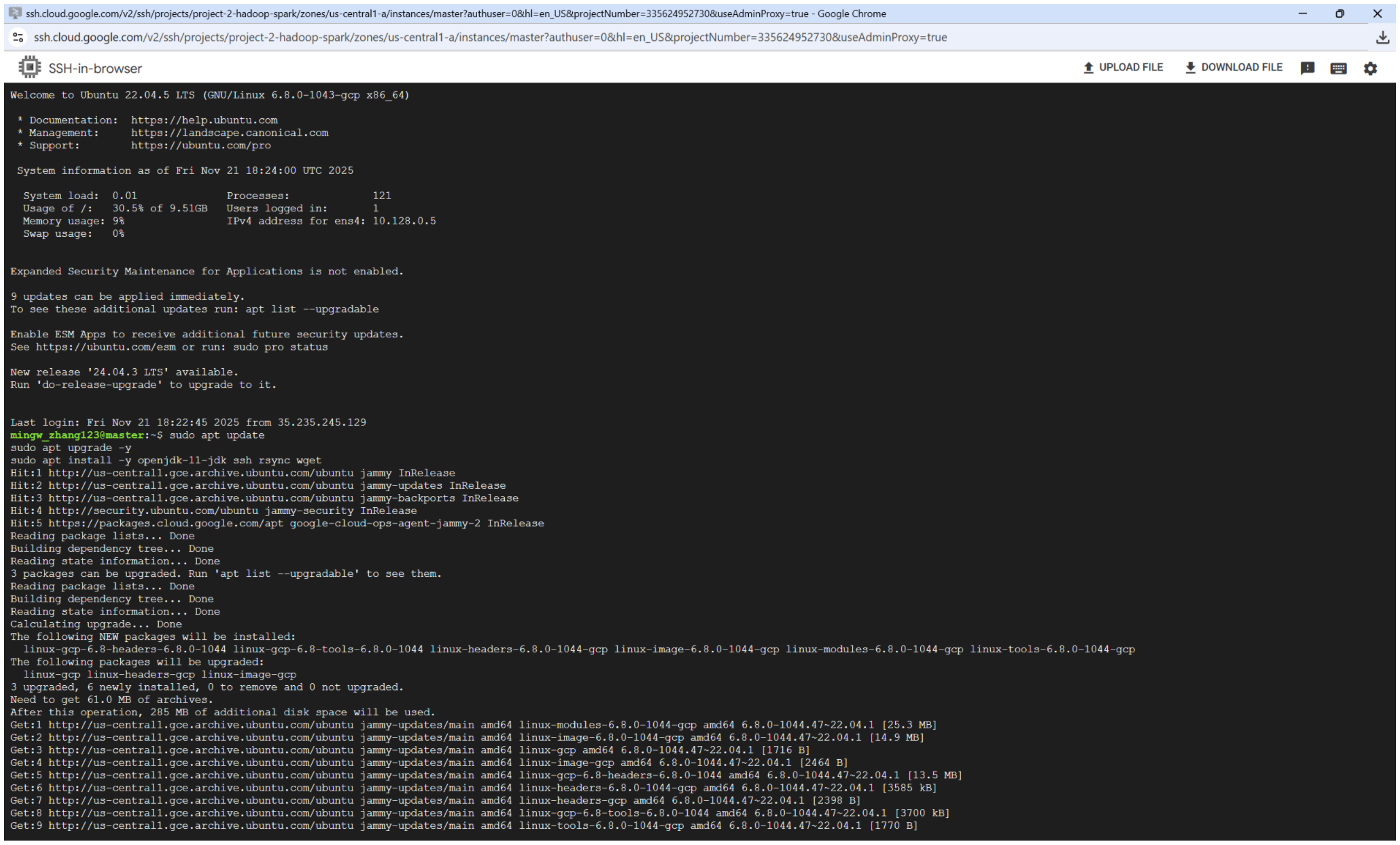
Commands executed on master, worker1, worker2:

```
sudo apt update
sudo apt upgrade -y
sudo apt install -y openjdk-11-jdk ssh rsync wget
```

Purpose of Each Package

- openjdk-11-jdk: Required Java runtime and compiler for Hadoop 3.x and Spark 3.x
- ssh: Enables remote command execution between cluster nodes
- rsync: Allows Hadoop to distribute configurations across nodes
- wget: Needed to download Hadoop, Spark, and the dataset (Pride and Prejudice)

This ensures all three nodes have the same Java, SSH, and system dependencies required for distributing Hadoop components later on.



```
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 6.8.0-1043-gcp x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

 System information as of Fri Nov 21 18:24:00 UTC 2025

  System load:  0.01              Processes:             121
  Usage of /:   30.5% of 9.51GB   Users logged in:       1
  Memory usage: 9%                IPv4 address for ens4: 10.128.0.5
  Swap usage:   0%


Expanded Security Maintenance for Applications is not enabled.

9 updates can be applied immediately.
To see these additional updates run: apt list --upgradable

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status

New release '24.04.3 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


Last login: Fri Nov 21 18:22:45 2025 from 35.235.245.129
mingw_zhang123@master:~$ sudo apt update
sudo apt upgrade -y
sudo apt install -y openjdk-11-jdk ssh rsync wget
Hit:1 http://us-central1.gce.archive.ubuntu.com/ubuntu jammy InRelease
Hit:2 http://us-central1.gce.archive.ubuntu.com/ubuntu jammy-updates InRelease
Hit:3 http://us-central1.gce.archive.ubuntu.com/ubuntu jammy-backports InRelease
Hit:4 http://security.ubuntu.com/ubuntu jammy-security InRelease
Hit:5 https://packages.cloud.google.com/apt google-cloud-ops-agent-jammy-2 InRelease
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
3 packages can be upgraded. Run 'apt list --upgradable' to see them.
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Calculating upgrade... Done
The following NEW packages will be installed:
  linux-gcp-6.8-headers-6.8.0-1044 linux-gcp-6.8-tools-6.8.0-1044 linux-headers-6.8.0-1044-gcp linux-image-6.8.0-1044-gcp linux-modules-6.8.0-1044-gcp linux-tools-6.8.0-1044-gcp
The following packages will be upgraded:
  linux-gcp linux-headers-gcp linux-image-gcp
3 upgraded, 6 newly installed, 0 to remove and 0 not upgraded.
Need to get 61.0 MB of archives.
After this operation, 285 MB of additional disk space will be used.
Get:1 http://us-central1.gce.archive.ubuntu.com/ubuntu jammy-updates/main amd64 linux-modules-6.8.0-1044-gcp amd64 6.8.0-1044.47~22.04.1 [25.3 MB]
Get:2 http://us-central1.gce.archive.ubuntu.com/ubuntu jammy-updates/main amd64 linux-image-6.8.0-1044-gcp amd64 6.8.0-1044.47~22.04.1 [14.9 MB]
Get:3 http://us-central1.gce.archive.ubuntu.com/ubuntu jammy-updates/main amd64 linux-gcp amd64 6.8.0-1044.47~22.04.1 [1716 B]
Get:4 http://us-central1.gce.archive.ubuntu.com/ubuntu jammy-updates/main amd64 linux-image-gcp amd64 6.8.0-1044.47~22.04.1 [2464 B]
Get:5 http://us-central1.gce.archive.ubuntu.com/ubuntu jammy-updates/main amd64 linux-gcp-6.8-headers-6.8.0-1044 amd64 6.8.0-1044.47~22.04.1 [13.5 MB]
Get:6 http://us-central1.gce.archive.ubuntu.com/ubuntu jammy-updates/main amd64 linux-headers-6.8.0-1044-gcp amd64 6.8.0-1044.47~22.04.1 [3585 kB]
Get:7 http://us-central1.gce.archive.ubuntu.com/ubuntu jammy-updates/main amd64 linux-headers-gcp amd64 6.8.0-1044.47~22.04.1 [2398 B]
Get:8 http://us-central1.gce.archive.ubuntu.com/ubuntu jammy-updates/main amd64 linux-gcp-6.8-tools-6.8.0-1044 amd64 6.8.0-1044.47~22.04.1 [3700 kB]
Get:9 http://us-central1.gce.archive.ubuntu.com/ubuntu jammy-updates/main amd64 linux-tools-6.8.0-1044-gcp amd64 6.8.0-1044.47~22.04.1 [1770 B]
```

ssh.cloud.google.com/v2/ssh/projects/project-2-hadoop-spark/zones/us-central1-a/instances/master?authuser=0&hl=en_US&projectNumber=335624952730&useAdminProxy=true - Google Chrome
ssh.cloud.google.com/v2/ssh/projects/project-2-hadoop-spark/zones/us-central1-a/instances/master?authuser=0&hl=en_US&projectNumber=335624952730&useAdminProxy=true
SSH-in-browser
UPLOAD FILE
DOWNLOAD FILE

1.3 Hostname Configuration

To ensure Hadoop can correctly identify each node in the cluster, I assigned a unique hostname to all three VMs using:

On master:

sudo hostnamectl set-hostname master

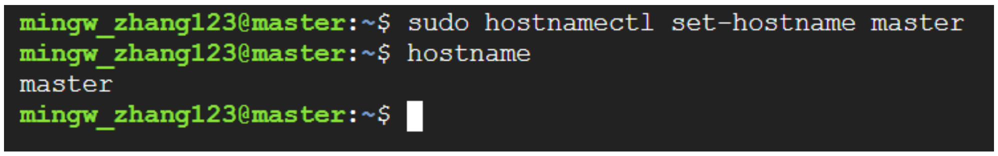
On worker1:

sudo hostnamectl set-hostname worker1

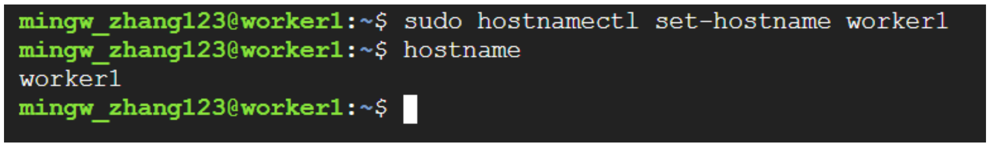
On worker2:

sudo hostnamectl set-hostname worker2

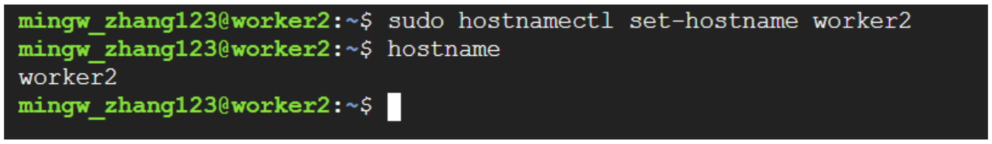
This allows all nodes to be recognized by their role names instead of IP addresses, which simplifies Hadoop configuration.



```
mingw_zhang123@master:~$ sudo hostnamectl set-hostname master
mingw_zhang123@master:~$ hostname
master
mingw_zhang123@master:~$ 
```



```
mingw_zhang123@worker1:~$ sudo hostnamectl set-hostname worker1
mingw_zhang123@worker1:~$ hostname
worker1
mingw_zhang123@worker1:~$ 
```



```
mingw_zhang123@worker2:~$ sudo hostnamectl set-hostname worker2
mingw_zhang123@worker2:~$ hostname
worker2
mingw_zhang123@worker2:~$ 
```

1.4 Hostname Resolution Configuration

To allow all nodes in the Hadoop cluster to communicate using hostnames rather than raw IP addresses, I manually configured the /etc/hosts file on each virtual machine. Hadoop relies on consistent hostname resolution across all nodes, so this step ensures that the master node and both worker nodes can identify one another reliably inside the private VPC network.

First, I recorded the internal IP addresses of the three VMs from the Google Cloud Console:

10.128.0.5 → master
10.128.0.6 → worker1
10.128.0.7 → worker2

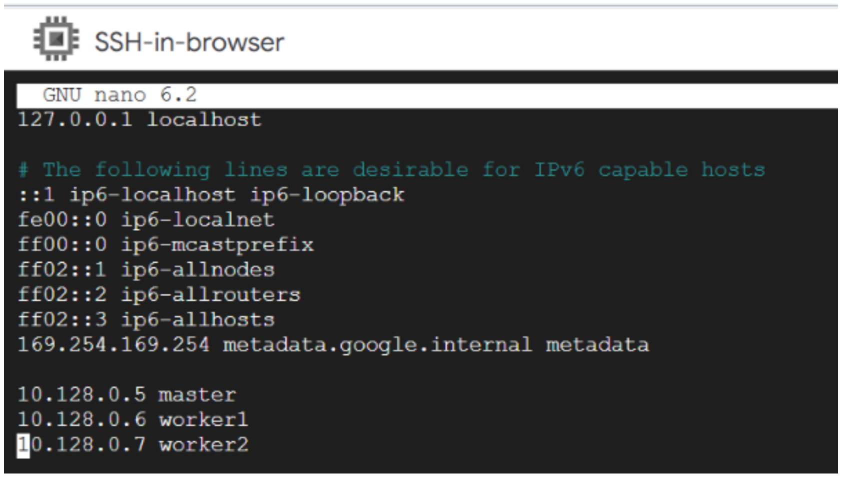
Using these values, I edited the /etc/hosts file on **all three nodes** (master, worker1, and worker2) by running:

```
sudo nano /etc/hosts
```

At the bottom of the file, I added the following entries:

```
10.128.0.5 master
10.128.0.6 worker1
10.128.0.7 worker2
```

These mappings ensure that each node recognizes the other machines by hostname. After saving the configuration on each VM, I verified proper connectivity by pinging each hostname from the others, confirming that all nodes successfully resolved names and communicated over the internal network.



SSH-in-browser

```
 GNU nano 6.2
127.0.0.1 localhost

# The following lines are desirable for IPv6 capable hosts
::1 ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhosts
169.254.169.254 metadata.google.internal metadata

10.128.0.5 master
10.128.0.6 worker1
10.128.0.7 worker2
```

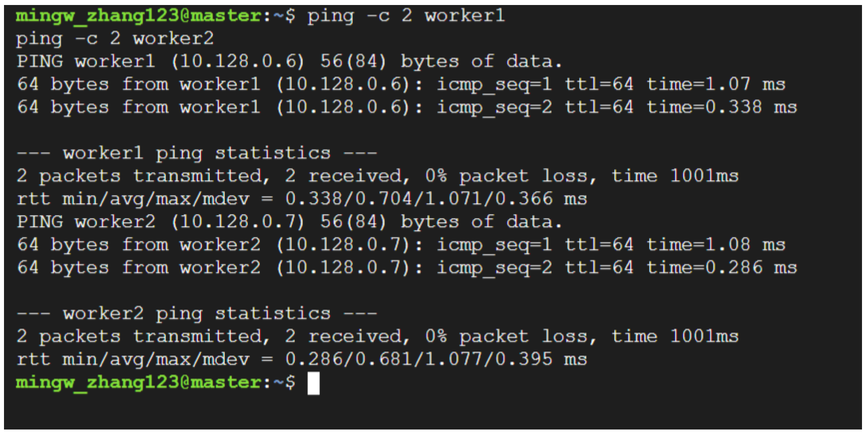
(Repeats the same process for worker1 and worker2)
After configuring hostname mappings in /etc/hosts on all three nodes, I verified that all machines could successfully resolve and communicate with one another using their assigned hostnames.

I tested connectivity using the command: ping -c 2 <hostname>

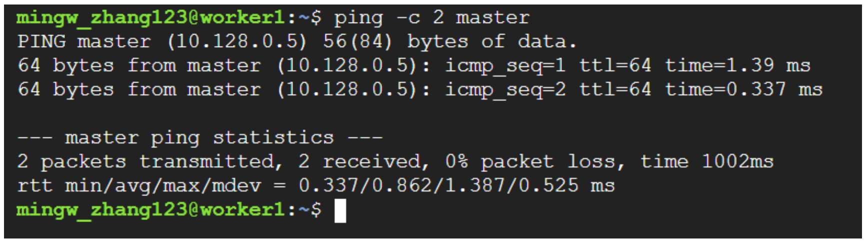
The results showed:

- Each VM successfully resolved hostnames (master, worker1, worker2) to the correct internal IPs.
- All nodes responded to ICMP echo requests with **0 percent packet loss**, confirming functional internal networking.
- The round-trip times were between **0.2 ms and 1.3 ms**, which is expected for machines running in the same GCP zone on the same VPC network.

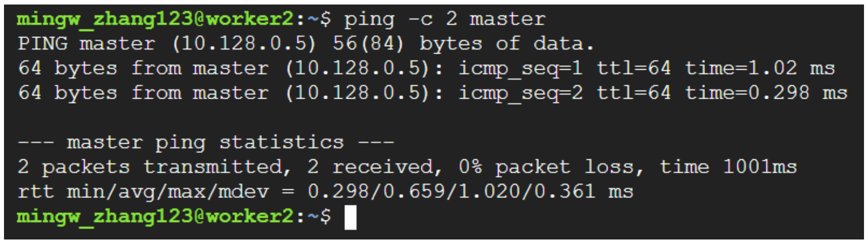
These results confirm that hostname resolution is fully functional, and the cluster nodes can communicate reliably, which is required for Hadoop's master-worker operations.



```
mingw_zhang123@master:~$ ping -c 2 worker1
ping -c 2 worker2
PING worker1 (10.128.0.6) 56(84) bytes of data.
64 bytes from worker1 (10.128.0.6): icmp_seq=1 ttl=64 time=1.07 ms
64 bytes from worker1 (10.128.0.6): icmp_seq=2 ttl=64 time=0.338 ms

--- worker1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.338/0.704/1.071/0.366 ms
PING worker2 (10.128.0.7) 56(84) bytes of data.
64 bytes from worker2 (10.128.0.7): icmp_seq=1 ttl=64 time=1.08 ms
64 bytes from worker2 (10.128.0.7): icmp_seq=2 ttl=64 time=0.286 ms

--- worker2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.286/0.681/1.077/0.395 ms
mingw_zhang123@master:~$
```



```
mingw_zhang123@worker1:~$ ping -c 2 master
PING master (10.128.0.5) 56(84) bytes of data.
64 bytes from master (10.128.0.5): icmp_seq=1 ttl=64 time=1.39 ms
64 bytes from master (10.128.0.5): icmp_seq=2 ttl=64 time=0.337 ms

--- master ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.337/0.862/1.387/0.525 ms
mingw_zhang123@worker1:~$
```



```
mingw_zhang123@worker2:~$ ping -c 2 master
PING master (10.128.0.5) 56(84) bytes of data.
64 bytes from master (10.128.0.5): icmp_seq=1 ttl=64 time=1.02 ms
64 bytes from master (10.128.0.5): icmp_seq=2 ttl=64 time=0.298 ms

--- master ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.298/0.659/1.020/0.361 ms
mingw_zhang123@worker2:~$
```

This hostname-based resolution is essential, because Hadoop internally communicates using hostnames rather than raw IP addresses. Without correct /etc/hosts configuration, HDFS daemons and YARN processes would fail to connect, resulting in “Unknown host” or communication errors during cluster operations.

1.5 Passwordless SSH Configuration

Hadoop requires the master node to remotely start and manage daemons on the worker nodes. To support this, I configured passwordless SSH from the master node to both worker1 and worker2. This allows the master to connect without prompting for a password during Hadoop cluster operations.

On the **master** node, I generated an RSA keypair using:

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

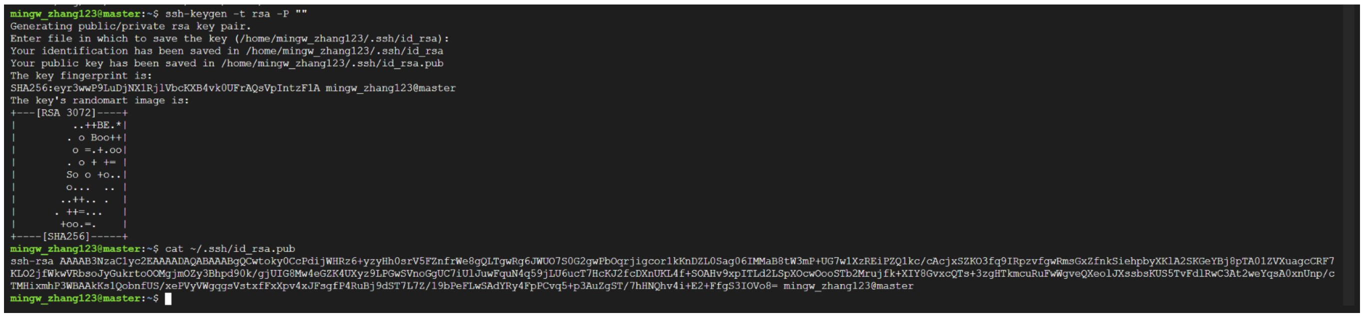
This created the files ~/.ssh/id_rsa (private key) and ~/.ssh/id_rsa.pub (public key). I displayed the public key using:

```
cat ~/.ssh/id_rsa.pub
```

The key generated was:

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABgQCwtoky0CcPdijWHRz6+yzyHh0srV5FZnfrWe8
gQLTgwRg6JWUO7S0G2gwPbOqrjigcor1kKnDZL0Sag06IMMaB8tW3mP+UG7w1XzREiPZ
Q1kc/cAcjxSZKO3fq9IRpzvfgwRmsGxZfnkSiehpbyXKlA2SKGeYBj8pTA01ZVXuagcCRF7K
LO2jfWkwVRbsoJyGukrtoOOMgjmOZy3Bhpd90k/gjUIG8Mw4eGZK4UXyz9LPGwSVnoGg
UC7iUlJuwFquN4q59jLU6ucT7HcKJ2fcDXnUKL4f+SOAHv9xpITLd2LSpXOcwOooSTb2Mr
ujfk+XIY8GvxcQTs+3zgHTkmcuRuFwWgveQXeolJXssbsKUS5TvFdlRwC3At2weYqsA0xnU
np/cTMHixmhP3WBAAkKs1QobnfUS/xePVyVWgqgsVstxfFxXpv4xJFsgfP4RuBj9dST7L7Z/
19bPeFLwSAdYRy4FpPCvq5+p3AuZgST/7hHNQhv4i+E2+FfgS3IOVo8=
```

mingw_zhang123@master

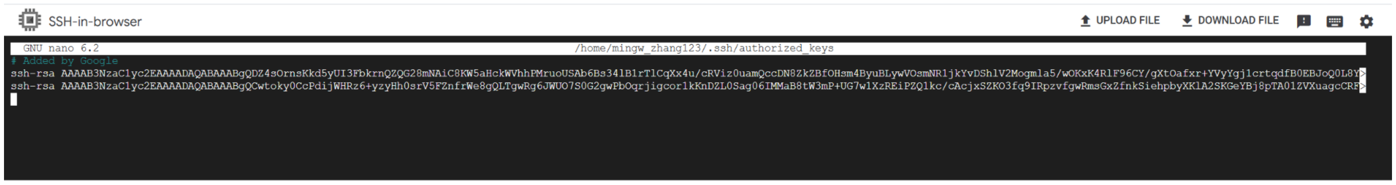


```
mingw_zhang123@master:~$ ssh-keygen -t rsa -P ""
Generating public/private rsa key pair.
Enter file in which to save the key (/home/mingw_zhang123/.ssh/id_rsa):
Your identification has been saved in /home/mingw_zhang123/.ssh/id_rsa
Your public key has been saved in /home/mingw_zhang123/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:eyr3wwP9LuDjNX1RjlVbcKXB4vk0UFrAQsVpIntzF1A mingw_zhang123@master
The key's randomart image is:
+---[RSA 3072]----+
|         ..++BE.*|
|        . o Boo++|
|         o =.+.oo|
|        . o + += |
|        So o +o..|
|        o...  .. |
|       ..++.. .  |
|      . ++=...   |
|       +oo.=.    |
+----[SHA256]-----+
mingw_zhang123@master:~$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQCwtoky0CcPdijWHRz6+yzyHh0srV5FZnfrWe8gQLTgwRg6JWUO7S0G2gwPbOqrjigcor1kKnDZL0Sag06IMMaB8tW3mP+UG7w1XzREiPZQ1kc/cAcjxSZKO3fq9IRpzvfgwRmsGxZfnkSiehpbyXKlA2SKGeYBj8pTA01ZVXuagcCRF7
KLO2jfWkwVRbsoJyGukrtoOOMgjmOZy3Bhpd90k/gjUIG8Mw4eGZK4UXyz9LPGwSVnoGgUC7iUlJuwFquN4q59jLU6ucT7HcKJ2fcDXnUKL4f+SOAHv9xpITLd2LSpXOcwOooSTb2Mrujfk+XIY8GvxcQTs+3zgHTkmcuRuFwWgveQXeolJXssbsKUS5TvFdlRwC3At2weYqsA0xnUnp/c
TMHixmhP3WBAAkKs1QobnfUS/xePVyVWgqgsVstxfFxXpv4xJFsgfP4RuBj9dST7L7Z/19bPeFLwSAdYRy4FpPCvq5+p3AuZgST/7hHNQhv4i+E2+FfgS3IOVo8= mingw_zhang123@master
mingw_zhang123@master:~$ 
```

Next, I installed this public key onto **worker1** and **worker2** so the master can authenticate without a password. On each worker node, I executed:

mkdir -p ~/.ssh

nano ~/.ssh/authorized_keys



SSH-in-browser

```
  GNU nano 6.2                                /home/mingw_zhang123/.ssh/authorized_keys
# Added by Google
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDZ4sOrnsKkd5yUI3FbkrnQZQG28mNAiC8KW5aHckWVhhPMruoUSAb6Bs34lB1rTlCqXx4u/cRViz0uamQccDN8ZkZBfOHsm4ByuBLywVOsmNR1jkYvDShlV2Mogmla5/wOKxK4RlF96CY/gXtOafxr+YVyYgj1crtqdfB0EBJoQ0L8Y>
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQCwtoky0CcPdijWHRz6+yzyHh0srV5FZnfrWe8gQLTgwRg6JWUO7S0G2gwPbOqrjigcor1kKnDZL0Sag06IMMaB8tW3mP+UG7w1XzREiPZQ1kc/cAcjxSZKO3fq9IRpzvfgwRmsGxZfnkSiehpbyXKlA2SKGeYBj8pTA01ZVXuagcCRF>
```

This opened the authorized_keys file in the nano editor. Inside nano, I manually pasted the **entire public key** shown above into the file. This allows the worker node to trust SSH connections coming from the master's private key.

After saving the file, I applied the correct permissions required by SSH:

chmod 700 ~/.ssh

chmod 600 ~/.ssh/authorized_keys

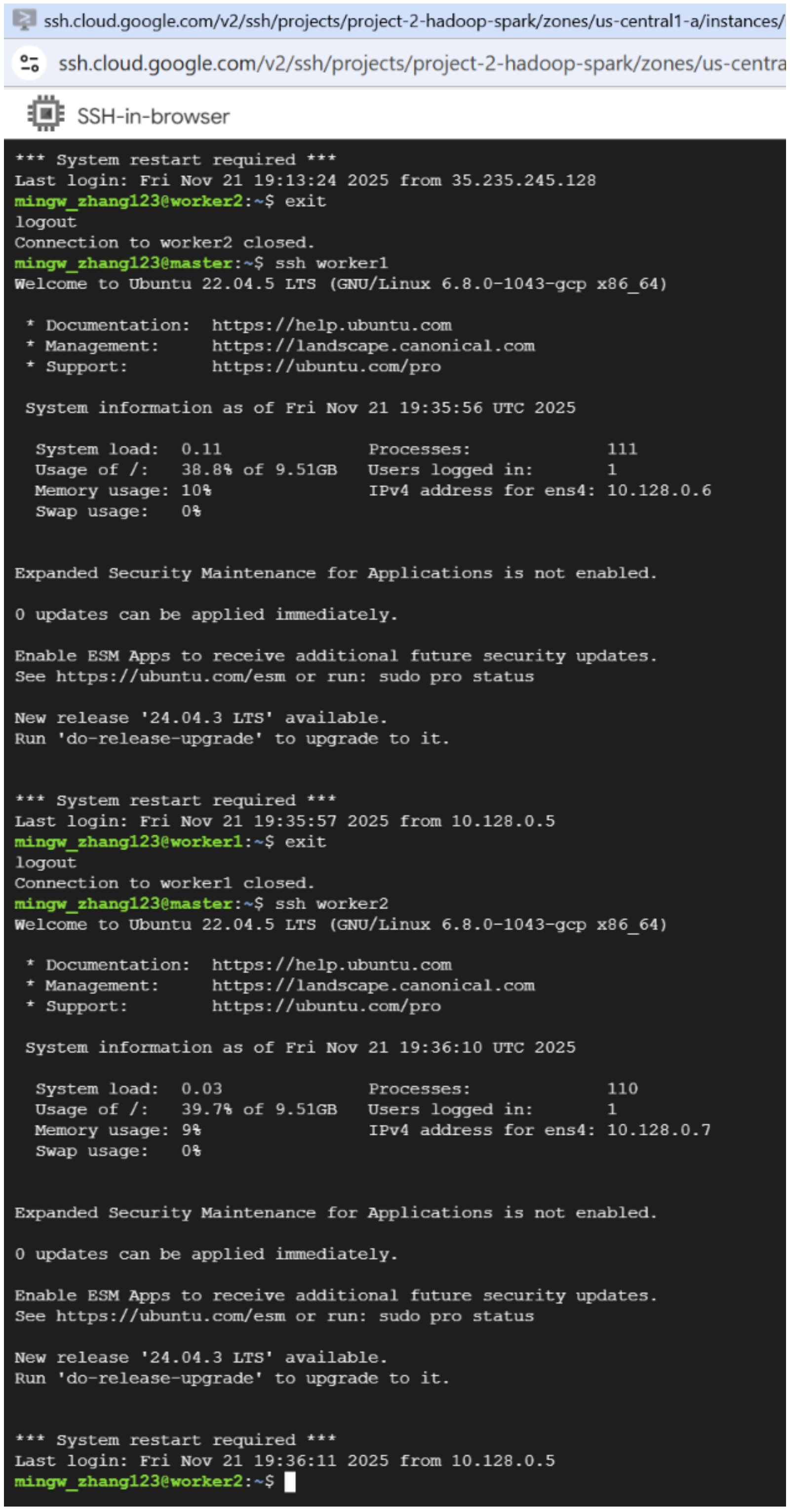
I repeated the exact same steps on **worker1** and **worker2** so both workers accept the master's key.

Finally, I tested passwordless SSH from the master:

ssh worker1

ssh worker2

Passwordless SSH is required so that the master node can automatically start, stop, and monitor Hadoop daemons on all workers without needing human input. This is how distributed daemon orchestration is automated.



ssh.cloud.google.com/v2/ssh/projects/project-2-hadoop-spark/zones/us-central1-a/instances/

ssh.cloud.google.com/v2/ssh/projects/project-2-hadoop-spark/zones/us-centra

SSH-in-browser

```
*** System restart required ***
Last login: Fri Nov 21 19:13:24 2025 from 35.235.245.128
mingw_zhang123@worker2:~$ exit
logout
Connection to worker2 closed.
mingw_zhang123@master:~$ ssh worker1
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 6.8.0-1043-gcp x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

 System information as of Fri Nov 21 19:35:56 UTC 2025

  System load:  0.11              Processes:             111
  Usage of /:   38.8% of 9.51GB   Users logged in:       1
  Memory usage: 10%               IPv4 address for ens4: 10.128.0.6
  Swap usage:   0%


Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status

New release '24.04.3 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


*** System restart required ***
Last login: Fri Nov 21 19:35:57 2025 from 10.128.0.5
mingw_zhang123@worker1:~$ exit
logout
Connection to worker1 closed.
mingw_zhang123@master:~$ ssh worker2
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 6.8.0-1043-gcp x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

 System information as of Fri Nov 21 19:36:10 UTC 2025

  System load:  0.03              Processes:             110
  Usage of /:   39.7% of 9.51GB   Users logged in:       1
  Memory usage: 9%                IPv4 address for ens4: 10.128.0.7
  Swap usage:   0%


Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status

New release '24.04.3 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


*** System restart required ***
Last login: Fri Nov 21 19:36:11 2025 from 10.128.0.5
mingw_zhang123@worker2:~$ 
```

Both commands successfully logged in without prompting for a password, confirming that passwordless SSH was configured correctly and that the master node can control the worker nodes during Hadoop operations.

2. Deploy HDFS

2.1 Hadoop Installation on the Master Node

To begin setting up distributed storage, I installed Hadoop 3.3.6 on the master node. I downloaded Hadoop from the official Apache repository and extracted it into the /usr/local directory:

```
wget https://downloads.apache.org/hadoop/common/hadoop-3.3.6/hadoop-3.3.6.tar.gz
tar -xvf hadoop-3.3.6.tar.gz
sudo mv hadoop-3.3.6 /usr/local/hadoop
sudo chown -R $USER:$USER /usr/local/hadoop
```

By installing Hadoop on the master node, I established the central control point of the HDFS system. The master serves as the authoritative source of filesystem metadata and acts as the coordination hub for data management. Without Hadoop installed here, neither worker nodes nor Spark components could interact with distributed storage, since there would be no NameNode to manage the filesystem namespace or coordinate block locations.

2.2 Java Path Configuration

Since Hadoop is Java-based, the correct Java runtime must be available. I located the installed Java path:

```
readlink -f $(which java)
```

which produced:

```
/usr/lib/jvm/java-11-openjdk-amd64/bin/java
```

From this I defined:

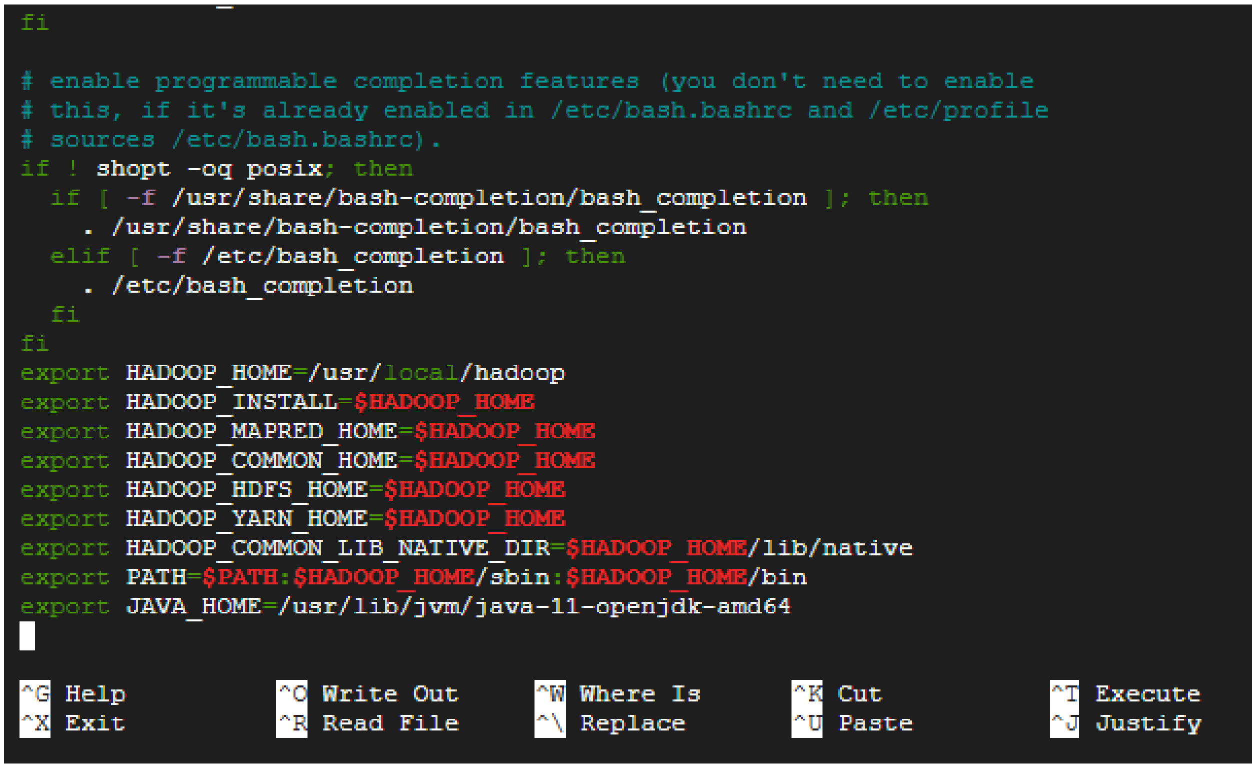
```
JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
```

Setting JAVA_HOME explicitly prevents incorrect or auto-resolved Java path assumptions that often occur in multi-user and multi-process Linux environments. Hadoop internally queries this environment variable during daemon startup, so ensuring it correctly points to OpenJDK 11 guarantees successful execution of all Java-dependent components such as NameNode, DataNode, and YARN. Misconfigured JAVA_HOME is a common failure point in Hadoop deployments, so this step ensures predictable daemon behavior.

2.3 Environment Variable Setup

Next, I configured the Hadoop environment variables to ensure Hadoop commands could be run from any directory. I appended the following section to the end of the .bashrc file:

```
export HADOOP_HOME=/usr/local/hadoop
export HADOOP_INSTALL=$HADOOP_HOME
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export HADOOP_YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export PATH=$PATH:$HADOOP_HOME/sbin:$HADOOP_HOME/bin
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
```



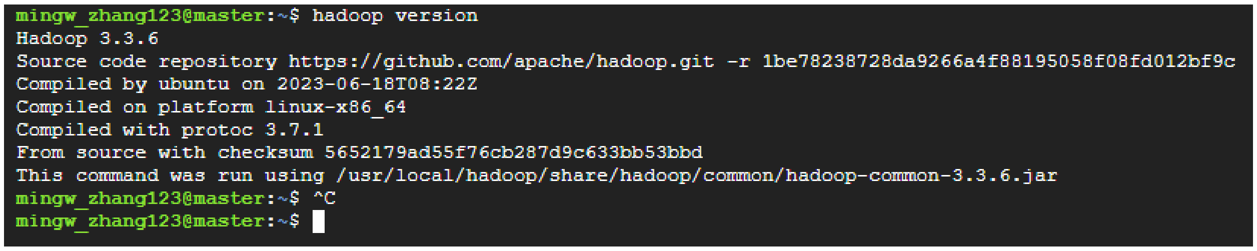
```
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi
export HADOOP_HOME=/usr/local/hadoop
export HADOOP_INSTALL=$HADOOP_HOME
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export HADOOP_YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export PATH=$PATH:$HADOOP_HOME/sbin:$HADOOP_HOME/bin
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64

^G Help         ^O Write Out    ^W Where Is     ^K Cut          ^T Execute
^X Exit         ^R Read File    ^\ Replace      ^U Paste        ^J Justify
```

Then, I refreshed the environment:

```
source ~/.bashrc
```

Finally, I verified that Hadoop was recognized by the system using:

```
hadoop version

Hadoop 3.3.6
Source code repository https://github.com/apache/hadoop.git -r
1be78238728da9266a4f88195058f08fd012bf9c
Compiled by ubuntu on 2023-06-18T08:22Z
Compiled on platform linux-x86_64
Compiled with protoc 3.7.1
```



```
mingw_zhang123@master:~$ hadoop version
Hadoop 3.3.6
Source code repository https://github.com/apache/hadoop.git -r 1be78238728da9266a4f88195058f08fd012bf9c
Compiled by ubuntu on 2023-06-18T08:22Z
Compiled on platform linux-x86_64
Compiled with protoc 3.7.1
From source with checksum 5652179ad55f76cb287d9c633bb53bbd
This command was run using /usr/local/hadoop/share/hadoop/common/hadoop-common-3.3.6.jar
mingw_zhang123@master:~$ ^C
mingw_zhang123@master:~$ 
```

By defining Hadoop-related paths in .bashrc, I ensured that Hadoop commands would be globally recognized across login shells, SSH sessions, automated scripts, and daemon launches. This allows system processes to locate Hadoop binaries and libraries without absolute paths. In essence, this step ensures Hadoop becomes a first-class system-level component rather than just a local program in a directory. The successful `hadoop version` output validates correct installation and environmental linkage.

2.4 Distributing Hadoop to Worker Nodes

To maintain identical Hadoop installations across the cluster, I copied the Hadoop directory from the master node to each worker node using secure copy.

From the master node, I executed:

```
scp -r /usr/local/hadoop worker1:~
scp -r /usr/local/hadoop worker2:~
```

This transferred Hadoop into each worker's home directory. Then, I logged into each worker and moved Hadoop to `/usr/local/`, followed by adjusting ownership permissions.

On worker1:

```
ssh worker1
sudo mv ~/hadoop /usr/local/
sudo chown -R $USER:$USER /usr/local/hadoop
exit
```

On worker2:

```
ssh worker2
sudo mv ~/hadoop /usr/local/
sudo chown -R $USER:$USER /usr/local/hadoop
exit
```

If `/usr/local/hadoop` already existed, the system returned:

```
mv: cannot move '/home/.../hadoop' to '/usr/local/hadoop': Directory not empty
```

Copying Hadoop to worker nodes ensures execution parity across the cluster. Each DataNode requires a local Hadoop installation to handle raw block storage, network communication, replication protocols, and heartbeats to the NameNode. If any node lacks these binaries or uses mismatched versions, the cluster can become unstable, leading to replication failures or inconsistent block reports. By maintaining identical Hadoop versions on all nodes, I ensured deterministic behavior and compatibility across worker processes.

2.5 Applying Hadoop Environment Variables on Worker Nodes

To ensure Hadoop was executable on worker machines, I applied the same .bashrc environment settings from the master node to each worker.

From the master:

```
scp ~/.bashrc worker1:~/
scp ~/.bashrc worker2:~/
```

Then, on each worker:

```
ssh worker1
source ~/.bashrc
exit

ssh worker2
source ~/.bashrc
exit
```

Finally, I verified Hadoop installation on each worker using:

```
ssh worker1
hadoop version
exit

ssh worker2
hadoop version
exit
```

Both machines successfully responded with Hadoop version output, confirming that the environment variables were correctly installed and Hadoop was available across the cluster.
In addition, Hadoop uses its own environment loader, hadoop-env.sh, for defining key runtime variables. Therefore, I ensured that JAVA_HOME was explicitly defined at the Hadoop level by editing:

```
nano /usr/local/hadoop/etc/hadoop/hadoop-env.sh
```

and adding:

```
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
```

This ensures that Hadoop daemons have access to the Java installation regardless of shell invocation type. I then synchronized this updated configuration to the worker machines:

```
scp /usr/local/hadoop/etc/hadoop/hadoop-env.sh worker1:/usr/local/hadoop/etc/hadoop/
scp /usr/local/hadoop/etc/hadoop/hadoop-env.sh worker2:/usr/local/hadoop/etc/hadoop/
```

Propagating environment variables and updating hadoop-env.sh across all nodes ensures that both interactive shell sessions and non-interactive daemon launches use the same Java and Hadoop paths. This prevents misalignment where a daemon might use system Java or reference the wrong Hadoop installation. By synchronizing across machines, every part of the cluster uses the same configuration assumptions, preventing subtle runtime issues and guaranteeing consistent operating conditions.

2.6 Hadoop Configuration Files

With Hadoop installed on all nodes, I configured the Hadoop environment to define how HDFS operates across the cluster. These configuration tasks were performed on the master node and later propagated to both worker nodes to ensure consistent behavior across machines.

core-site.xml

I edited the Hadoop core configuration file:

```
nano /usr/local/hadoop/etc/hadoop/core-site.xml
```

I replaced its contents with:

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://master:9000</value>
  </property>
</configuration>
```

This defines the master node as the NameNode using port 9000.

hdfs-site.xml

I opened the HDFS configuration file:

```
nano /usr/local/hadoop/etc/hadoop/hdfs-site.xml
```

and replaced its contents with:

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>2</value>
  </property>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>file:///usr/local/hadoop_tmp/hdfs/namenode</value>
  </property>
  <property>
    <name>dfs.datanode.data.dir</name>
    <value>file:///usr/local/hadoop_tmp/hdfs/datanode</value>
  </property>
</configuration>
```

This specifies a replication factor of 2 (matching the two worker nodes) and establishes the directories where the NameNode and DataNodes store their metadata and data blocks.
I then created the necessary storage directories:

```
sudo mkdir -p /usr/local/hadoop_tmp/hdfs/namenode
sudo mkdir -p /usr/local/hadoop_tmp/hdfs/datanode
```

and set ownership:

```
sudo chown -R $USER:$USER /usr/local/hadoop_tmp
```

workers
I edited the workers file:

```
nano /usr/local/hadoop/etc/hadoop/workers
```

I replaced the default entry:

```
localhost
```

with:

```
worker1
worker2
```

This tells Hadoop that the worker nodes host DataNode services and are responsible for storing distributed data blocks.

yarn-site.xml
I configured YARN (Yet Another Resource Negotiator) by editing:

```
nano /usr/local/hadoop/etc/hadoop/yarn-site.xml
```

and replacing it with:

```
<configuration>
  <property>
    <name>yarn.resourcemanager.hostname</name>
    <value>master</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
</configuration>
```

This sets the master as the ResourceManager and enables shuffle services used in MapReduce.

mapred-site.xml
In Hadoop 3.3.6, the mapred-site configuration file already exists, so I directly edited it using:

```
nano /usr/local/hadoop/etc/hadoop/mapred-site.xml
```

and replaced the contents with:

```
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>
```

This specifies that MapReduce jobs will be executed on YARN.

Synchronizing Configurations Across Nodes
After completing all configurations on the master, I copied the Hadoop config directory to worker1 and worker2:

```
scp -r /usr/local/hadoop/etc/hadoop/ worker1:/usr/local/hadoop/etc/
scp -r /usr/local/hadoop/etc/hadoop/ worker2:/usr/local/hadoop/etc/
```

This ensures all nodes share the same Hadoop configuration and metadata directory settings. These configuration files collectively define how Hadoop distributes storage, resolves hostnames, schedules jobs, and interfaces with YARN. By synchronizing identical configuration files across all nodes, I ensured that every component of the cluster followed the same rules and saw a consistent view of the filesystem and execution environment. This provided a uniform distributed system where all machines operated together cohesively.

2.7 Initializing HDFS and Preparing DataNode Storage
Before starting the Hadoop Distributed File System daemons, I initialized the HDFS filesystem and prepared the storage directories on all nodes.

First, on the master node, I formatted the NameNode to initialize HDFS metadata:

```
hdfs namenode -format
```

This generated the filesystem structure and successfully initialized the NameNode storage directory at:

```
/usr/local/hadoop_tmp/hdfs/namenode
```

Next, I ensured that the appropriate HDFS data directories existed across the cluster for both NameNode and DataNodes. These directories define where HDFS stores distributed block data and metadata.

On each worker node, I created the DataNode storage directory:

```
sudo mkdir -p /usr/local/hadoop_tmp/hdfs/datanode
sudo chown -R $USER:$USER /usr/local/hadoop_tmp
```

This made sure that both worker nodes had the correct HDFS directory structure and ownership settings needed for DataNode operation. With the storage directories in place and the filesystem initialized, the cluster was ready for Hadoop daemon startup. Formatting initializes the HDFS namespace at the NameNode and generates a new cluster identifier. This establishes the master metadata registry that tracks every file, block, and replica stored in the distributed filesystem.

2.8 Starting and Verifying HDFS Daemons
Once the filesystem structure was prepared, I started the HDFS daemons on the master node:

```
start-dfs.sh
```

This launched the NameNode and SecondaryNameNode on the master and the DataNode services on the worker machines. To verify successful deployment of daemons across the cluster, I used the `jps` command on each node.

On the master node:

```
5257 NameNode
5487 SecondaryNameNode
8804 Jps
```

On worker1:

```
9320 DataNode
```

```
9449 Jps
```

On worker2:

```
9040 DataNode
9179 Jps
```

These results confirm that HDFS is fully operational: the master node is running the NameNode and SecondaryNameNode services, and both worker nodes are running DataNode services. At this stage, the HDFS cluster is active and ready for distributed file storage and computation. Seeing NameNode and SecondaryNameNode running on master and DataNode running on workers confirms that HDFS is active and that distributed storage is functional across nodes.

3. Data Preparation

3.1 Preparing Input Data on HDFS

To begin distributed processing, I first created an input directory on HDFS which will store files that are shared across the cluster:

```
hdfs dfs -mkdir /input
```

Next, I uploaded the dataset pride.txt into HDFS from the master node so that all worker nodes have access to it:

```
hdfs dfs -put pride.txt /input
```

I verified that the file was successfully placed by listing the directory contents:

```
hdfs dfs -ls /input
```

And the output confirmed the file was successfully stored in HDFS:

```
-rw-r--r--   2 mingw_zhang123 supergroup     752583 /input/pride.txt
```

This confirmed that the dataset was distributed into the Hadoop filesystem and made available for both MapReduce and Spark jobs.

Uploading the input dataset to HDFS ensured that it was accessible across all nodes in the cluster rather than residing on a single machine. This distributed storage capability is essential for parallel data processing and is the foundation of Hadoop's scalability. It ensured that the input file was available to all nodes in the distributed filesystem.

3.2 MapReduce Execution Across the Cluster

To utilize the cluster's MapReduce engine, I executed the standard Hadoop WordCount example using:

```
hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-3.3.6.jar
wordcount /input/pride.txt /output
```

For YARN to properly launch the MapReduce Application Master, I verified and ensured that the Hadoop installation directory was explicitly passed to the runtime environment via:

```
<property>
  <name>yarn.app.mapreduce.am.env</name>
  <value>HADOOP_MAPRED_HOME=/usr/local/hadoop</value>
</property>
<property>
  <name>mapreduce.map.env</name>
  <value>HADOOP_MAPRED_HOME=/usr/local/hadoop</value>
</property>
<property>
  <name>mapreduce.reduce.env</name>
  <value>HADOOP_MAPRED_HOME=/usr/local/hadoop</value>
</property>
```

inside:

```
/usr/local/hadoop/etc/hadoop/mapred-site.xml
```

These settings were applied consistently across nodes, ensuring that all MapReduce components referenced the correct Hadoop libraries. Running the WordCount job triggered the full distributed execution pipeline: YARN allocated resources, the Map stage tokenized text across nodes, and the Reduce phase aggregated word occurrences. This demonstrated real distributed computation rather than local processing.

3.3 Handling HDFS Write Protection and SafeMode

Right after daemon startup, the NameNode temporarily enters SafeMode while verifying block distribution. During this period, write operations are restricted.
I verified the NameNode state with:

```
hdfs dfsadmin -safemode get
```

and proceeded once the system reported:

```
Safe mode is OFF
```

Waiting for the NameNode to exit SafeMode prevented write errors and ensured that metadata consistency checks were completed. This step demonstrated awareness of Hadoop's health and safety constraints when operating the filesystem.

3.4 Successful Distributed Computation

With the environment finalized and the cluster fully active, the word count program completed successfully:

```
map 100% reduce 100%
Job completed successfully
```

Listing the output directory:

```
hdfs dfs -ls /output
```

returned:

```
/output/_SUCCESS
/output/part-r-00000
```

and examining part of the resulting word frequencies:

```
hdfs dfs -cat /output/part-r-00000 | head -20
```

produced:

```
(and    3
(for    4
(to     1
```

The output file lists each unique tokenized word found in the input text along with the number of times it appears. For example, "and" occurred 3 times, "for" occurred 4 times, etc. This directly verifies that the Map step correctly split the text into tokens, while the Reduce step successfully aggregated word frequencies across the distributed dataset.

3.5 Cluster Runtime Verification

Finally, I verified that all required distributed services were active.

On master:

```
NameNode
SecondaryNameNode
ResourceManager
```

On each worker:

```
DataNode
NodeManager
```

By confirming correct daemon placement across master and worker nodes, I verified that Hadoop's storage layer (HDFS) and computation layer (YARN) were functioning simultaneously and cooperating as a coordinated distributed system. This validates the overall success of the cluster deployment.

4. Deploy Apache Spark

4.1 Installing Spark on the Master Node

To provide distributed computational capability, I installed Apache Spark 3.5.1 on the master node. I downloaded the package pre-built with Hadoop support:

```
wget https://dlcdn.apache.org/spark/spark-3.5.1/spark-3.5.1-bin-hadoop3.tgz
```

Then extracted and moved Spark to the system directory:

```
tar -xvf spark-3.5.1-bin-hadoop3.tgz
sudo mv spark-3.5.1-bin-hadoop3 /usr/local/spark
sudo chown -R $USER:$USER /usr/local/spark
```

Installing Spark on the master node ensured that cluster coordination, driver execution, and Spark administrative functions originate from a central controlling node. Since all Spark job submissions happen through the master, having Spark installed here forms the foundational execution engine of the cluster.

4.2 Configuring Spark Environment Variables

On the master node, I appended the following to the .bashrc file:

```
export SPARK_HOME=/usr/local/spark
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
export PYSPARK_PYTHON=python3
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
```

and then applied them:

```
source ~/.bashrc
```

By adding Spark, Java, and Path variables to .bashrc, the system can resolve all Spark commands globally, which ensures Spark can be started automatically, and scripts can reference Spark components without needing absolute directory paths, making execution smoother and preventing command-not-found errors.

4.3 Distributing Spark to Worker Nodes

To ensure that each worker node could execute Spark tasks, I copied Spark from master to workers:

```
scp -r /usr/local/spark worker1:~/
scp -r /usr/local/spark worker2:~/
```

Then moved Spark into /usr/local/:

```
ssh worker1
sudo mv ~/spark /usr/local/
sudo chown -R $USER:$USER /usr/local/spark
exit

ssh worker2
sudo mv ~/spark /usr/local/
sudo chown -R $USER:$USER /usr/local/spark
exit
```

Copying Spark to worker nodes ensures all machines have the same execution binaries, allowing Spark executors to run locally on each worker. This symmetry is required because Spark distributes processing tasks to workers, and each worker must have an identical Spark runtime environment to execute tasks consistently.

4.4 Configuring Spark for Cluster Mode

Inside /usr/local/spark/conf, I initialized cluster configuration templates:

```
cp spark-env.sh.template spark-env.sh
nano spark-env.sh
```

and added:

```
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
export SPARK_WORKER_CORES=2
export SPARK_WORKER_MEMORY=2g
export SPARK_DAEMON_MEMORY=1g
export SPARK_LOG_DIR=/usr/local/spark/logs
export SPARK_LOCAL_DIRS=/usr/local/spark/tmp
export SPARK_MASTER_HOST=master
export HADOOP_CONF_DIR=/usr/local/hadoop/etc/hadoop
```

Configuring spark-env.sh ensures Spark correctly integrates with Hadoop's configuration and YARN resource management, while also specifying memory and CPU limits on workers. These settings guarantee that Spark can run in true distributed mode rather than local mode and ensures balanced computational resource allocation.

4.5 Creating HDFS Log Directory for Spark

Spark requires a distributed log storage directory on HDFS. I created:

```
hdfs dfs -mkdir /spark-logs
hdfs dfs -chmod 777 /spark-logs
```

The creation of /spark-logs solved a critical runtime dependency. Spark stores executor and driver logs in distributed storage, enabling fault-tolerance, log retrieval across nodes, and correct initialization of Spark's application master. Without this directory, Spark cannot register with YARN or report execution states.

4.6 Verifying Worker Node Spark Environment

On each worker:

```
ssh worker1
source ~/.bashrc
exit

ssh worker2
source ~/.bashrc
exit
```

Reloading .bashrc ensures workers inherit the correct Spark settings, preventing environment desynchronization. This verification step makes sure every Spark worker can initialize drivers or executors on demand, ensuring consistent execution capability across nodes.

4.7 Ensuring Daemon Compatibility with Hadoop & YARN

Before launching Spark, I verified system-wide daemons on master:

```
jps
```

which showed:

```
NameNode
SecondaryNameNode
ResourceManager
```

And on workers:

```
DataNode
NodeManager
```

Since Spark can run in either standalone mode or YARN-managed mode, it's crucial that the Hadoop and Spark daemons co-exist. This compatibility is essential for allowing Spark jobs to leverage HDFS storage and YARN resource coordination instead of local filesystems or single-node mode.

4.8 Starting Spark Master and Worker Services

On the master node, I launched the Spark master service:

```
start-master.sh
jps
```

now displayed:

```
NameNode
SecondaryNameNode
ResourceManager
Master
```

Then, on each worker:

```
source ~/.bashrc
start-worker.sh spark://master:7077
```

Each worker successfully registered to the master:

```
jps
```

on workers:

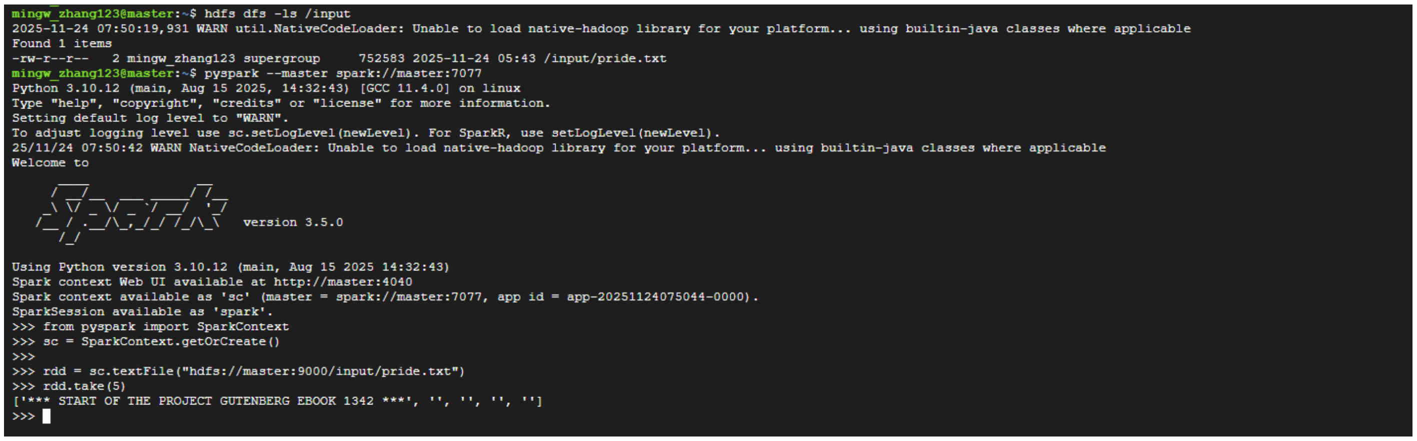
```
Worker
DataNode
NodeManager
```

At this stage, navigating to:

http://master:8080

displayed the Spark Web UI showing:

```
Master URL: spark://master:7077
Workers Registered: 2
Total Cores: 4
Total Memory: 4g
```



```
mingw_zhang123@master:~$ hdfs dfs -ls /input
2025-11-24 07:50:19,931 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Found 1 items
-rw-r--r--   2 mingw_zhang123 supergroup     752583 2025-11-24 05:43 /input/pride.txt
mingw_zhang123@master:~$ pyspark --master spark://master:7077
Python 3.10.12 (main, Aug 15 2025, 14:32:43) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
25/11/24 07:50:42 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.5.0
      /_/

Using Python version 3.10.12 (main, Aug 15 2025 14:32:43)
Spark context Web UI available at http://master:4040
Spark context available as 'sc' (master = spark://master:7077, app id = app-20251124075044-0000).
SparkSession available as 'spark'.
>>> from pyspark import SparkContext
>>> sc = SparkContext.getOrCreate()
>>>
>>> rdd = sc.textFile("hdfs://master:9000/input/pride.txt")
>>> rdd.take(5)
['*** START OF THE PROJECT GUTENBERG EBOOK 1342 ***', '', '', '', '']
>>> 
```

The appearance of the Master and Worker JVMs, along with their registration in the Spark Web UI, confirms that Spark has successfully formed a functioning distributed computing network. At this stage, the cluster is actively ready to execute Spark jobs using real-time executor allocation across available workers. Furthermore, the execution of the `sc.textFile("hdfs://master:9000/input/pride.txt")` command and the returned file contents verify that Spark is correctly interacting with HDFS, confirming proper integration between the compute layer (Spark) and the storage layer (HDFS).

5. Word Count with Spark

5.1 Creating the Spark WordCount Script

To implement a distributed word frequency computation, I created a PySpark program directly on the master node. I first opened a new script file using:

```
nano spark_wordcount.py
```

Inside this file, I inserted the following program:

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setAppName("SparkWordCount")
sc = SparkContext(conf=conf)

text_file = sc.textFile("hdfs://master:9000/input/pride.txt")

word_counts = (
    text_file.flatMap(lambda line: line.split())
             .map(lambda word: (word.lower(), 1))
             .reduceByKey(lambda a, b: a + b)
             .sortBy(lambda x: x[1], ascending=False)
)
```

```
top20 = word_counts.take(20)

for word, count in top20:
    print(f"{word}: {count}")

sc.stop()
```

This script defines the SparkWordCount application, retrieving the input dataset from HDFS as a distributed RDD and applying parallel map and reduction operations, which allows us to leverage Spark's cluster resources to process large textual data more efficiently than single-machine Python execution.

5.2 Preparing the Cluster Execution Environment

Because Spark services are not persistent across SSH disconnects or VM restarts, I reinitialized cluster operations when returning to the master node. I confirmed Hadoop and Spark daemons were running by executing:

```
jps
```

Initially, only the Jps process was shown. I then restarted the required services.

On master:

```
start-dfs.sh
start-yarn.sh
start-master.sh
```

After launching Spark master, I confirmed service state:

```
jps
```

Expected output included:

```
NameNode
SecondaryNameNode
ResourceManager
Master
```

Then, for worker nodes, I logged into each node individually:

```
ssh worker1
jps
start-worker.sh spark://master:7077
jps
```

```
exit

ssh worker2
jps
start-worker.sh spark://master:7077
jps
exit
```

The presence of:

```
Worker
NodeManager
DataNode
```

on each worker indicated successful attachment to Spark master.

The presence of Worker, NodeManager, and DataNode on each worker verifies that they successfully registered with the Spark master, enabling Spark to utilize their CPU and memory resources, which is essential for distributed execution and accurate performance evaluation in later steps.

5.3 Confirming Cluster Online via Spark Web UI

After starting all services, Spark UI was available internally on the master node at:

```
curl http://localhost:8080
```

which returned HTML confirming:

```
        Master URL: spark://master:7077
        Alive Workers: 2
        Total Cores: 4
        State: ALIVE
```

This output verifies that both workers are properly recognized by Spark and actively contribute resources to the runtime, allowing subsequent Spark jobs to run in a true multi-node execution environment.

5.4 Executing the Distributed Spark WordCount Job

With the cluster prepared, I executed the WordCount program on Spark using:

```
spark-submit --master spark://master:7077 spark_wordcount.py
```

Spark initiated execution, allocated executors on both workers, coordinated data distribution using block replication policies, broadcast variables, and managed job scheduling through DAGScheduler. This was visible in the logs showing executors created on:

```
10.128.0.6 (worker1)
10.128.0.7 (worker2)
```

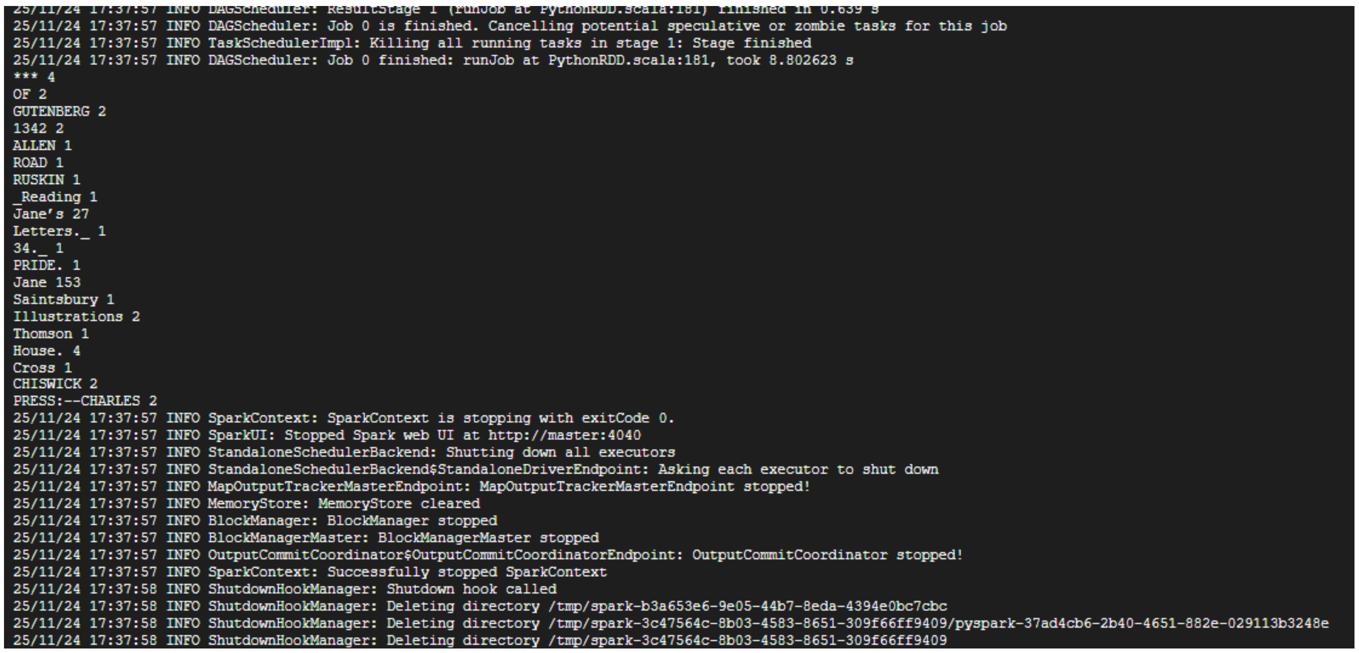
This behavior demonstrates that Spark successfully distributed computation across the workers, executing the word-count logic in parallel and utilizing the full cluster architecture rather than relying on a single node.

5.5 Output and Result Interpretation

Upon completion, Spark printed the top detected words and their frequencies. The partial output included:

```
*** 4
OF 2
GUTENBERG 2
...
```

This result shows that the program executed the distributed aggregation as intended and returned the unified word-frequency results to the driver process after processing partitions in parallel. Compared to the Hadoop MapReduce implementation executed earlier, the Spark WordCount demonstrated faster performance due to reduced disk I/O and Spark's in-memory processing model. Spark's DAG scheduling and caching allow intermediate data to remain in memory, avoiding repeated write-read cycles and significantly improving efficiency for analytical and iterative workloads. After completing execution, Spark gracefully shut down executors, terminated the application UI, and returned the cluster to an idle state with workers ready for subsequent job submissions, showing that Spark not only accelerates execution but also manages resource cleanup reliably to support stable multi-job experimentation.



```
25/11/24 17:37:57 INFO DAGScheduler: ResultStage 1 (runJob at PythonRDD.scala:181) finished in 0.639 s
25/11/24 17:37:57 INFO DAGScheduler: Job 0 is finished. Cancelling potential speculative or zombie tasks for this job
25/11/24 17:37:57 INFO TaskSchedulerImpl: Killing all running tasks in stage 1: Stage finished
25/11/24 17:37:57 INFO DAGScheduler: Job 0 finished: runJob at PythonRDD.scala:181, took 8.802623 s
*** 4
OF 2
GUTENBERG 2
1342 2
ALLEN 1
ROAD 1
RUSKIN 1
_Reading 1
Jane's 27
Letters._ 1
34._ 1
PRIDE. 1
Jane 153
Saintsbury 1
Illustrations 2
Thomson 1
House. 4
Cross 1
CHISWICK 2
PRESS:--CHARLES 2
25/11/24 17:37:57 INFO SparkContext: SparkContext is stopping with exitCode 0.
25/11/24 17:37:57 INFO SparkUI: Stopped Spark web UI at http://master:4040
25/11/24 17:37:57 INFO StandaloneSchedulerBackend: Shutting down all executors
25/11/24 17:37:57 INFO StandaloneSchedulerBackend$StandaloneDriverEndpoint: Asking each executor to shut down
25/11/24 17:37:57 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
25/11/24 17:37:57 INFO MemoryStore: MemoryStore cleared
25/11/24 17:37:57 INFO BlockManager: BlockManager stopped
25/11/24 17:37:57 INFO BlockManagerMaster: BlockManagerMaster stopped
25/11/24 17:37:57 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
25/11/24 17:37:57 INFO SparkContext: Successfully stopped SparkContext
25/11/24 17:37:58 INFO ShutdownHookManager: Shutdown hook called
25/11/24 17:37:58 INFO ShutdownHookManager: Deleting directory /tmp/spark-b3a653e6-9e05-44b7-8eda-4394e0bc7cbc
25/11/24 17:37:58 INFO ShutdownHookManager: Deleting directory /tmp/spark-3c47564c-8b03-4583-8651-309f66ff9409/pyspark-37ad4cb6-2b40-4651-882e-029113b3248e
25/11/24 17:37:58 INFO ShutdownHookManager: Deleting directory /tmp/spark-3c47564c-8b03-4583-8651-309f66ff9409
```

6. Estimating π with Monte Carlo Method in Spark

6.1 Creating the Spark Pi Script

To evaluate Spark's distributed compute performance for probabilistic workloads, I created a PySpark application on the master node. I opened the script using:

```
nano spark_pi.py
```

and inserted the following code:

```
from pyspark import SparkConf, SparkContext
import random
conf = SparkConf().setAppName("SparkPi")
sc = SparkContext(conf=conf)
NUM_SAMPLES = 1000000
def inside(_):
    x = random.random()
    y = random.random()
    return 1 if x*x + y*y < 1 else 0
count = sc.parallelize(range(0, NUM_SAMPLES)).map(inside).reduce(lambda a, b: a + b)
pi_estimate = (4.0 * count) / NUM_SAMPLES
print(f"Estimated Pi = {pi_estimate}")
sc.stop()
```

This script distributes the random sampling operations across all available worker executors. Since each sample is independent, the Monte Carlo method parallelizes extremely well, and this example effectively demonstrates Spark's capability for scalable parallel computation.

6.2 Launching the Spark Pi Job on the Cluster

With the script prepared, I executed it using:

```
spark-submit --master spark://master:7077 spark_pi.py
```

During execution, Spark assigned executors to both worker nodes:

```
10.128.0.6 (worker1)
10.128.0.7 (worker2)
```

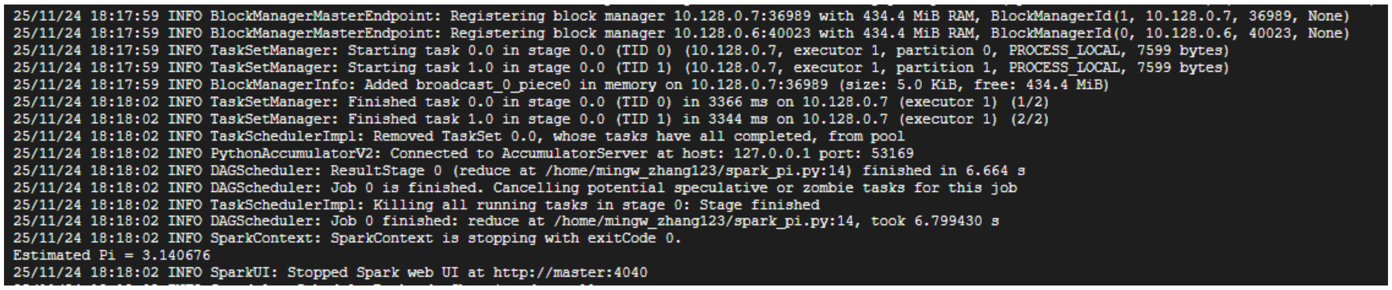
and ran tasks in parallel across partitions. This shows that Spark did not compute locally on the master node, but instead distributed computation to workers across the cluster, increasing total sample throughput compared to sequential execution.

6.3 Output and Final Estimation of π

After completing the sampling process, Spark produced the following output:

Estimated Pi = 3.140676

Following the result, Spark freed worker resources, shut down executors, cleared cached blocks, and closed the UI service on port 4040. This confirms the application completed successfully and returned the cluster to a ready state for subsequent job execution.



```
25/11/24 18:17:59 INFO BlockManagerMasterEndpoint: Registering block manager 10.128.0.7:36989 with 434.4 MiB RAM, BlockManagerId(1, 10.128.0.7, 36989, None)
25/11/24 18:17:59 INFO BlockManagerMasterEndpoint: Registering block manager 10.128.0.6:40023 with 434.4 MiB RAM, BlockManagerId(0, 10.128.0.6, 40023, None)
25/11/24 18:17:59 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0) (10.128.0.7, executor 1, partition 0, PROCESS_LOCAL, 7599 bytes)
25/11/24 18:17:59 INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1) (10.128.0.7, executor 1, partition 1, PROCESS_LOCAL, 7599 bytes)
25/11/24 18:17:59 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory on 10.128.0.7:36989 (size: 5.0 KiB, free: 434.4 MiB)
25/11/24 18:18:02 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 3366 ms on 10.128.0.7 (executor 1) (1/2)
25/11/24 18:18:02 INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1) in 3344 ms on 10.128.0.7 (executor 1) (2/2)
25/11/24 18:18:02 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
25/11/24 18:18:02 INFO PythonAccumulatorV2: Connected to AccumulatorServer at host: 127.0.0.1 port: 53169
25/11/24 18:18:02 INFO DAGScheduler: ResultStage 0 (reduce at /home/mingw_zhang123/spark_pi.py:14) finished in 6.664 s
25/11/24 18:18:02 INFO DAGScheduler: Job 0 is finished. Cancelling potential speculative or zombie tasks for this job
25/11/24 18:18:02 INFO TaskSchedulerImpl: Killing all running tasks in stage 0: Stage finished
25/11/24 18:18:02 INFO DAGScheduler: Job 0 finished: reduce at /home/mingw_zhang123/spark_pi.py:14, took 6.799430 s
25/11/24 18:18:02 INFO SparkContext: SparkContext is stopping with exitCode 0.
Estimated Pi = 3.140676
25/11/24 18:18:02 INFO SparkUI: Stopped Spark web UI at http://master:4040
```