

Analysis of mortgage cost with an Apache Spark cluster

Roger Ferrod, University of Turin

This tutorial aims to replicate a Bloomberg analysis¹ concerning mortgage costs in the United States by distributing the computation workload over a small Apache-Spark cluster deployed on Chameleon. The study is focused on the disparity between racial groups and how it is exacerbated by non-bank lenders. Data is retrieved from the Home Mortgage Disclosure Act database², a comprehensive collection of the mortgage maker activities that lender providers report to their regulators. For this study, some multiple linear regression analyses are needed, as well as a preprocessing phase aimed at filtering and cleaning the data.

Even though the dataset size may not require distributed computation across multiple nodes, the deployment of a small cluster can be seen as an exercise and preparation for scaling up the project. Moreover, even with a single node environment, Spark provides a boost in performance, if compared to native Python or R scripts, thanks to its built-in parallelization and optimization engine.

The cluster setup will consist of three virtual machines managed by OpenStack: one master node and two workers. For distributing the data across the nodes and coordinating the jobs, the Hadoop Distributed File System (HDFS) and YARN will be used. The architecture is summarized in Figure 1.

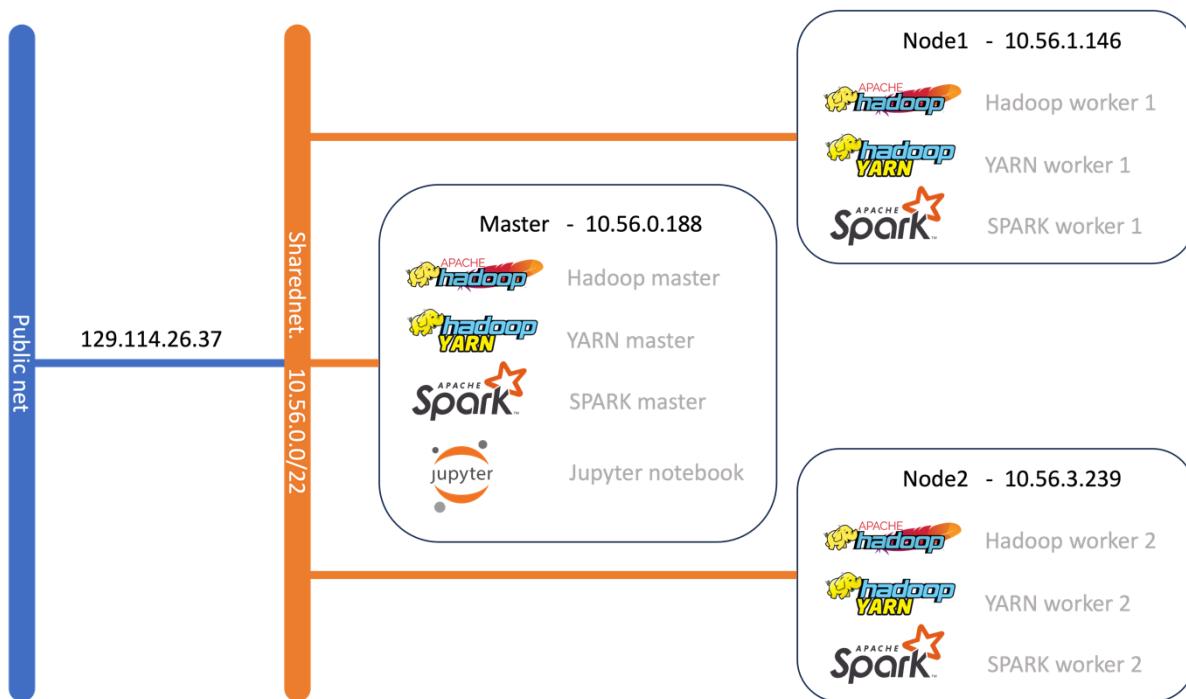


Figure 1

¹ <https://www.bloomberg.com/graphics/2023-nonbank-lender-mortgage-loan-borrower-fee/>

² <https://ffiec.cfpb.gov/>

Create and initialize the virtual machines

- As a first step, we create the master node by setting the instance name (Figure 2 a), the Ubuntu 22.04 source image (Figure 2 b), *m1.medium* flavor (Figure 2 c), a shared private network (Figure 2 d) and our SSH public key for accessing the VM once launched (Figure 3).

a)

b)

c)

d)

Figure 2

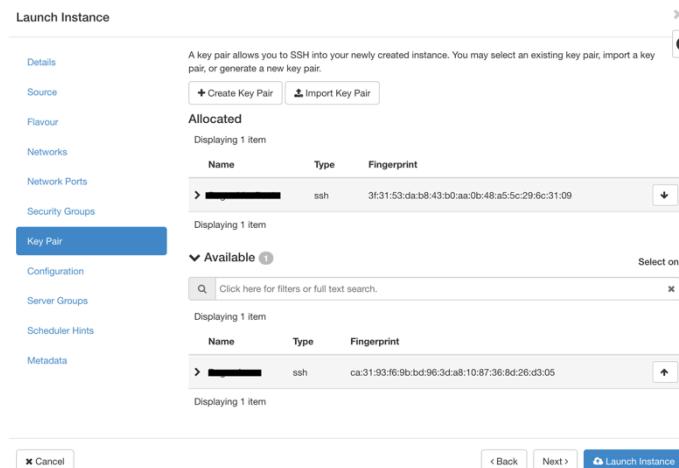


Figure 3

We also provide an *init-script* (Figure 4) that will set up the environment. More in detail, the script will:

- 1) create a user named *hadoop*
- 2) install Java, Apache-Hadoop and Apache-Spark
- 3) set up the environment variables

```

#!/bin/bash
sudo useradd -m hadoop
echo hadoop:hadoop | sudo chpasswd
sudo usermod -aG hadoop hadoop
sudo adduser hadoop sudo
sudo chsh -s /bin/bash hadoop

sudo apt update
sudo apt install default-jdk -y
cd /usr/lib/jvm/
sudo ln -sf java-11-openjdk* current-java

cd /home/cc
wget https://dlcdn.apache.org/hadoop/common/hadoop-3.3.6/hadoop-3.3.6.tar.gz
tar -xzf hadoop-3.3.6.tar.gz
rm hadoop-3.3.6.tar.gz
sudo mv hadoop-3.3.6 /opt/
sudo ln -sf /opt/hadoop-3.3.6 /opt/hadoop
sudo chown hadoop:root /opt/hadoop* -R
sudo chmod g+rwx /opt/hadoop* -R
cd /home/cc
wget https://dlcdn.apache.org/spark/spark-3.4.2/spark-3.4.2-bin-hadoop3.tgz
tar xvf spark-3.4.2-bin-hadoop3.tgz
rm spark-3.4.2-bin-hadoop3.tgz
sudo mv spark-3.4.2-bin-hadoop3 /opt/
sudo ln -sf /opt/spark-3.4.2-bin-hadoop3 /opt/spark
sudo chown hadoop:root /opt/spark* -R
sudo chmod g+rwx /opt/spark* -R

echo '
export JAVA_HOME=/usr/lib/jvm/current-java
export HADOOP_HOME="/opt/hadoop"
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export HADOOP_HDFS_HOME=$HADOOP_HOME
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_YARN_HOME=$HADOOP_HOME
export SPARK_HOME=/opt/spark
export PATH=$PATH:/opt/hadoop/bin:/opt/hadoop/sbin:/opt/spark/bin
' >> /home/hadoop/.bashrc

```

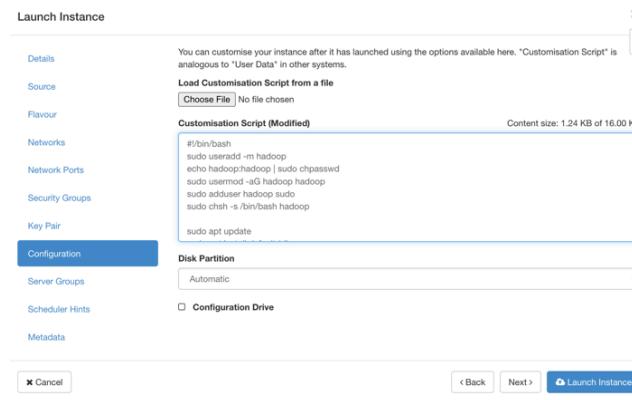


Figure 4

- 2) Meanwhile, we can allocate a public IP (Figure 5 a) and associate it with the instance (Figure 5 b) in order to reach the master node from the external network.

Figure 5

- 3) We also create a new volume to be able to store the dataset that exceeds the capacity of a *m1.medium* instance. 50 GB will be sufficient for our purposes (Figure 6 a). Once created we attach it to the master node (Figure 6 b).

Figure 6

Figure 7 summarizes the VM details, including the public IP and private IP (internal shared network).

PySpark-master

Overview	Interfaces	Log	Console	Action Log
Name PySpark-master ID 106d38df-3f30-4718-946b-6b97dc562938 Description - Project ID eb94788f04c047dfb4c797bef9f0fb7e Status Active Locked False Availability Zone nova Created 25 Mar 2024, 12:19 a.m. Age 3 minutes Host c08-24				
Specs				
Flavour Name m1.medium Flavour ID 3 RAM 4GB VCPUs 2 VCPU Disk 40GB				
IP Addresses				
sharednet1 10.56.0.188, 129.114.26.37				
Security Groups				
default ALLOW IPv4 icmp from 0.0.0.0/0 ALLOW IPv4 22/tcp from 0.0.0.0/0 ALLOW IPv6 to ::/0 ALLOW IPv4 to 0.0.0.0/0 ALLOW IPv6 from default ALLOW IPv4 80/tcp from 0.0.0.0/0 ALLOW IPv4 9000-9010/tcp from 0.0.0.0/0 ALLOW IPv4 from default				
Metadata				
Key Name Roger-MacBook Image Name CC-Ubuntu22.04 Image ID c2a51044-f3fd-4eac-8e69-f58fa4f90109				
Volumes Attached				
Attached To pyspark-data on /dev/vdb				

Figure 7

- 4) First, once connected to the VM through the `ssh cc@129.114.26.37` command, we create an ssh key pair and import the public key to OpenStack (Figure 8).

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

Import Public Key

Key Pair Name *

pyspark-master

Key Type *

Load Public Key from a file

Choose File No file chosen

Public Key * (Modified)

Content size: 570 bytes of 16.00 KB

```
ssh-rsa AAAAB3NzaC1yc2EAAAQABAA...  
...  
U= cc@pyspark-master|
```

Cancel **Import Public Key**

Figure 8

- 5) At this point, we can create and launch two virtual machines that will serve as workers. From the OpenStack web interface, we set the number of desired instances to 2 (Figure 9) and we repeat the same steps done for the master node, except for the key pair configuration, where we select the newly imported key of the master node. We also take note of the private IP associated with the new VMs (Figure 10).

Launch Instance

Details

Please provide the initial hostname for the instance, the availability zone where it will be deployed, and the instance count. Increase the Count to create multiple instances with the same settings.

Instance Name * PySpark-workers

Flavour * m1.medium

Networks

Network Ports

Availability Zone nova

Security Groups

Key Pair

Count * 2

Configuration

Server Groups

Scheduler Hints

Metadata

Total Instances (160 Max)
39%
61 Current Usage
2 Added
97 Remaining

Cancel **Next >** **Launch Instance**

Figure 9

Instances

Displaying 20 items

Instance Name	Image Name	IP Address	Flavour	Key Pair	Status	Availability Zone	Task	Power State	Age	Actions
PySpark-workers-2	CC-Ubuntu22.04	10.56.3.239	m1.medium	pyspark-master	Build	nova	Spawning	No State	0 minutes	<button>Associate Floating IP</button>
PySpark-workers-1	CC-Ubuntu22.04	10.56.1.146	m1.medium	pyspark-master	Build	nova	Spawning	No State	0 minutes	<button>Associate Floating IP</button>
PySpark-master	CC-Ubuntu22.04	10.56.0.188, 129.114.26.37	m1.medium	Roger-MacBook	Active	nova	None	Running	8 minutes	<button>Create Snapshot</button>

Figure 10

- 6) From the master console, we begin the host configuration with the following commands:

```
sudo hostnamectl set-hostname master  
sudo nano /etc/hosts
```

inside the /etc/hosts file we add the following lines:

```
10.56.0.188 master  
10.56.1.146 node1  
10.56.3.239 node2
```

we also delete (or comment) the localhost entry and IPv6 addresses.

- 7) We repeat the same process by connecting to the two workers, but in that case, only a reference to the master node is needed:

```
ssh cc@node1  
sudo hostnamectl set-hostname node1  
sudo nano /etc/hosts  
    10.56.0.188 master  
exit  
  
ssh cc@node2  
sudo hostnamectl set-hostname node2  
sudo nano /etc/hosts  
    10.56.0.188 master  
exit
```

- 8) Similarly, we create an *ssh key-pair* for also the Hadoop user, we copy the master's public key and paste it on the worker's nodes.

```
su hadoop
cd ../hadoop
ssh-keygen
cat .ssh/id_rsa.pub

sudo su cc
ssh cc@node1
su hadoop
cd /home/hadoop
ssh-keygen
nano .ssh/authorized_keys
exit
exit

ssh cc@node2
su hadoop
cd /home/hadoop
ssh-keygen
nano .ssh/authorized_keys
exit
exit
```

Configuring Hadoop, YARN and Spark

- 1) Back again to the Hadoop user on the master node, we begin configuring Apache-Hadoop by editing its configuration files. We add the following lines to the \$HADOOP_HOME/etc/hadoop/core-site.xml file as shown in Figure 11

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

<!-- Put site-specific property overrides in this file. --&gt;

&lt;configuration&gt;
    &lt;property&gt;
        &lt;name&gt;fs.default.name&lt;/name&gt;
        &lt;value&gt;hdfs://master:9000&lt;/value&gt;
    &lt;/property&gt;
&lt;/configuration&gt;</pre>
```

Figure 11

Similarly, for the \$HADOOP_HOME/etc/hadoop/hdfs-site.xml file, we add:

```
<configuration>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>/opt/hadoop/data/nameNode</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>/opt/hadoop/data/dataNode</value>
    </property>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.webhdfs.enabled</name>
        <value>true</value>
    </property>
    <property>
        <name>dfs.permissions</name>
        <value>false</value>
    </property>
</configuration>
```

We also add the line

```
export JAVA_HOME=/usr/lib/jvm/current-java
```

to the \$HADOOP_HOME/etc/hadoop/hadoop-env.sh script

and finally, we list the nodes that will be used as workers with the following command:

```
echo 'node1
node2' > $HADOOP_HOME/etc/hadoop/workers
```

- 2) From the master node, we copy the configuration files to the worker's nodes with the commands:

```
scp $HADOOP_HOME/etc/hadoop/* node1:$HADOOP_HOME/etc/hadoop/
scp $HADOOP_HOME/etc/hadoop/* node2:$HADOOP_HOME/etc/hadoop/
```

- 3) We continue the Hadoop configuration by adding the Hadoop user public key
`cat /home/hadoop/.ssh/id_rsa.pub >> /home/hadoop/.ssh/authorized_keys`

and by editing the \$HADOOP_HOME/etc/hadoop/yarn-site.xml file with the following lines:

```
<configuration>
    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>master</value>
    </property>
</configuration>
```

As before we copy the files to the worker's nodes:

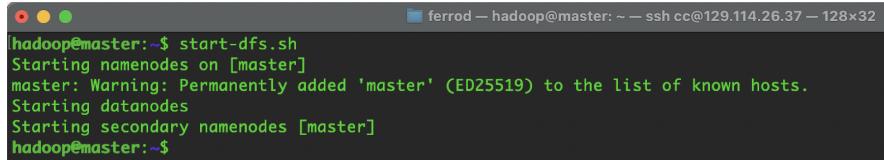
```
scp $HADOOP_HOME/etc/hadoop/yarn-site.xml node1:$HADOOP_HOME/etc/hadoop/yarn-site.xml
```

```
scp $HADOOP_HOME/etc/hadoop/yarn-site.xml node2:$HADOOP_HOME/etc/hadoop/yarn-site.xml
```

- 4) As usually done with common file systems, also HDFS needs to be formatted. We execute:

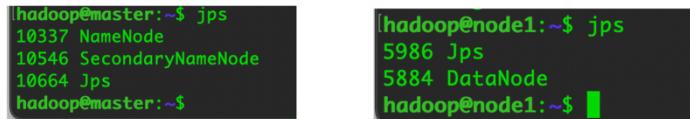
```
hdfs namenode -format  
  
ssh hadoop@node1  
hdfs namenode -format  
exit  
  
ssh hadoop@node2  
hdfs namenode -format  
exit
```

- 5) From the master node, with the `start-dfs.sh` command, we launch the Hadoop service on the cluster (Figure 12). We can have immediate feedback with the `jps` command both on master (Figure 13 a) and worker's nodes (Figure 13 b).



```
ferrod — hadoop@master: ~ — ssh cc@129.114.26.37 — 128x32  
hadoop@master:~$ start-dfs.sh  
Starting namenodes on [master]  
master: Warning: Permanently added 'master' (ED25519) to the list of known hosts.  
Starting datanodes  
Starting secondary namenodes [master]  
hadoop@master:~$
```

Figure 12



```
hadoop@master:~$ jps  
10337 NameNode  
10546 SecondaryNameNode  
10664 Jps  
hadoop@master:~$  
  
hadoop@node1:~$ jps  
5986 Jps  
5884 DataNode  
hadoop@node1:~$
```

a)

b)

Figure 13

More details can be found by connecting to the Hadoop interface at 127.0.0.1:9870. For this purpose, an ssh-tunnel can be opened, from the local host, with:

```
ssh cc@129.114.26.37 -L 9870:127.0.0.1:9870 -N
```

The result will be similar to what is shown in Figure 14. Please notice the number of living nodes, which in our case should be 2. In the Datanodes section, we can monitor the disk usage of worker nodes (Figure 15).

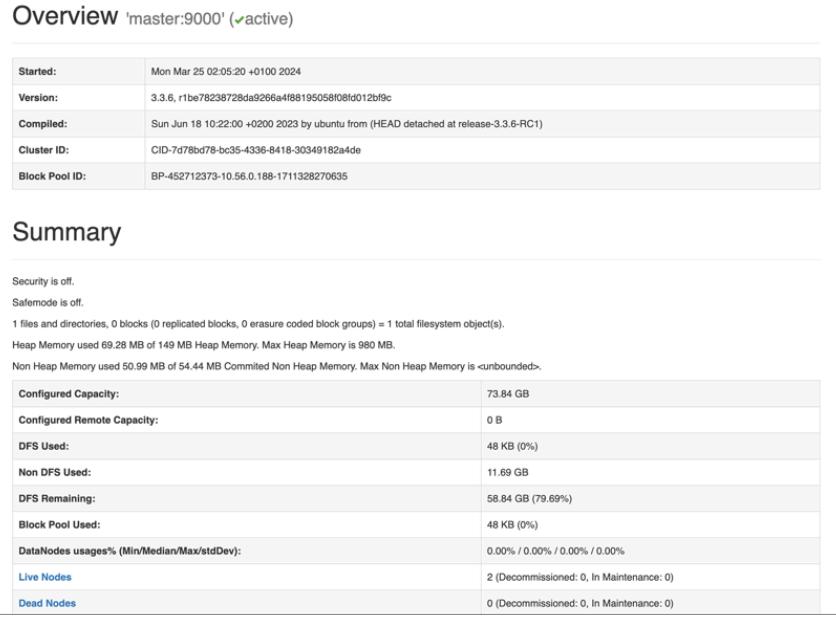


Figure 14

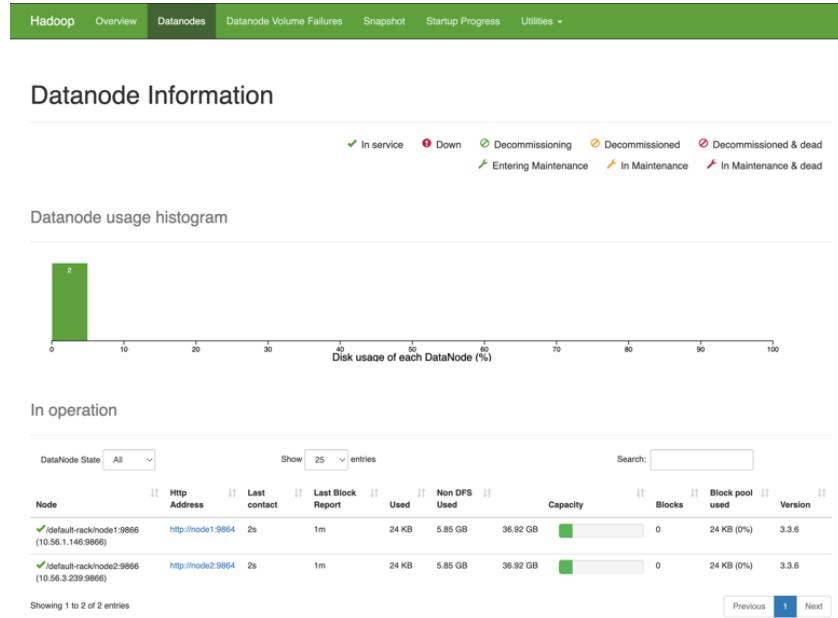


Figure 15

- 6) YARN is launched with the `start-yarn.sh` script and can be monitored from the 8088 port. The `yarn node -list` command will provide a list of working nodes; the output should be similar to Figure 16.

```
[hadoop@master:~$ yarn node --list
2024-03-25 01:07:12,968 INFO client.DefaultNoHARMFailoverProxyProvider: Connecting to ResourceManager at master/10.56.0.188:8032
Total Nodes:
  Node-Id          Node-State Node-Http-Address      Number-of-Running-Containers
  node2:37799      RUNNING    node2:8042                  0
  node1:44177      RUNNING    node1:8042                  0
hadoop@master:~$ ]
```

Figure 16

- 7) Finally, we set up Apache-Spark by editing its configuration files, more in detail we copy the spark-env.sh template with

```
mv $SPARK_HOME/conf/spark-env.sh.template $SPARK_HOME/conf/spark-env.sh
```

and add to the \$SPARK_HOME/conf/spark-env.sh script the following lines:

```
export SPARK_MASTER_HOST=master
export JAVA_HOME=/usr/lib/jvm/current-java
```

We also write the list of workers in the \$SPARK_HOME/conf/workers file:

```
node1
node2
```

- 8) The Spark cluster can be launched with the \$SPARK_HOME/sbin/start-all.sh command (Figure 17) and monitored at port 8080. The interface will look like Figure 18, where we can see 2 active workers.

```
[hadoop@master:~$ $SPARK_HOME/sbin/start-all.sh
starting org.apache.spark.deploy.master.Master, logging to /opt/spark/logs/spark-hadoop-org.apache.spark.deploy.master.Master-1
master.out
node2: starting org.apache.spark.deploy.worker.Worker, logging to /opt/spark/logs/spark-hadoop-org.apache.spark.deploy.worker.Worker-1-node2.out
node1: starting org.apache.spark.deploy.worker.Worker, logging to /opt/spark/logs/spark-hadoop-org.apache.spark.deploy.worker.Worker-1-node1.out
hadoop@master:~$
```

Figure 17

The screenshot shows the Apache Spark web UI at spark://master:7077. The top navigation bar includes links for Overview, Executors, Applications, and History. The main content area is titled "Spark Master at spark://master:7077". It provides summary statistics: URL: spark://master:7077, Alive Workers: 2, Cores in use: 4 Total, 0 Used, Memory in use: 5.6 GB Total, 0.0 B Used, Resources in use: 0 Running, 0 Completed, Drivers: 0 Running, 0 Completed, and Status: ALIVE. Below this, there are three expandable sections: "Workers (2)", "Running Applications (0)", and "Completed Applications (0)". The "Workers (2)" section lists two workers with their addresses, states, cores, and memory usage. The "Running Applications (0)" and "Completed Applications (0)" sections are currently empty.

Worker Id	Address	State	Cores	Memory	Resources
worker-20240325010947-10.56.1.146-37167	10.56.1.146:37167	ALIVE	2 (0 Used)	2.8 GB (0.0 B Used)	
worker-20240325010947-10.56.3.239-46863	10.56.3.239:46863	ALIVE	2 (0 Used)	2.8 GB (0.0 B Used)	

Figure 18

Prepare the dataset

- 1) Since the 40GB of storage provided by the master node is not sufficient to collect the entire dataset, we have attached an empty 50GB volume to the VM, as we can see from the `lsblk` command (Figure 19).

```
hadoop@master:~$ lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
loop0    7:0     0  63.9M  1 loop /snap/core20/2105
loop1    7:1     0  40.4M  1 loop /snap/snapd/20671
loop2    7:2     0 111.9M  1 loop /snap/lxd/24322
vda    252:0     0   40G  0 disk 
└─vda1 252:1     0   550M  0 part /boot/efi
└─vda2 252:2     0     8M  0 part
└─vda3 252:3     0   39.5G 0 part /
vdb    252:16    0    50G  0 disk
```

Figure 19

- 2) We now proceed to format and mount it with the following commands:

```
sudo mkfs -t ext4 /dev/vdb
cd /home/hadoop
mkdir hdd
sudo mount /dev/vdb hdd
sudo chown hadoop hdd
sudo chgrp hadoop hdd
```

We also make the mount point permanent by editing the `/etc/fstab` file with the following line (Figure 20)

```
/dev/vdb /home/cc/hdd ext4 defaults 0 0
```

```
GNU nano 6.2                                     /etc/fstab
LABEL=cloudimg-rootfs / ext4 defaults 0 1
LABEL=MKFS_ESP /boot/efi vfat defaults 0 2
/dev/vdb /home/cc/hdd ext4 defaults 0 0
```

Figure 20

- 3) Inside the `hdd` volume we can now store the dataset. For our analysis, we both need the collection of *Loan/Application Records (LAR)* and the *Reporter Panel*, which contains institutional information of the lender providers. We download the zipped csv files into two separate subfolders:

```
cd hdd

mkdir data panel

wget -P ./data
    https://s3.amazonaws.com/cfpb-hmda-public/prod/three-year-data/2018/2018_public_lar_three_year_cs
    v.zip
    https://s3.amazonaws.com/cfpb-hmda-public/prod/three-year-data/2019/2019_public_lar_three_year_cs
    v.zip
    https://s3.amazonaws.com/cfpb-hmda-public/prod/one-year-data/2020/2020_public_lar_one_year_csv.zi
    p
    https://s3.amazonaws.com/cfpb-hmda-public/prod/one-year-data/2021/2021_public_lar_one_year_csv.zi
    p https://s3.amazonaws.com/cfpb-hmda-public/prod/snapshot-data/2022/2022_public_lar_csv.zip

wget -P ./panel
    https://s3.amazonaws.com/cfpb-hmda-public/prod/three-year-data/2018/2018_public_panel_three_year_
    csv.zip
    https://s3.amazonaws.com/cfpb-hmda-public/prod/three-year-data/2019/2019_public_panel_three_year_
    csv.zip
    https://s3.amazonaws.com/cfpb-hmda-public/prod/one-year-data/2020/2020_public_panel_one_year_cs
    v.zip
    https://s3.amazonaws.com/cfpb-hmda-public/prod/one-year-data/2021/2021_public_panel_one_year_cs
    v.zip https://s3.amazonaws.com/cfpb-hmda-public/prod/snapshot-data/2022/2022_public_panel_csv.zip
```

- 4) If absent, we install the unzip package with `sudo apt install unzip` and proceed by deflating the zip files:

```
unzip "./data/*.zip" -d ./data
unzip "./panel/*.zip" -d ./panel
```

once decompressed we can delete the original zip files:

```
rm ./data/*.zip
rm ./panel/*.zip
```

- 5) The dataset is now entirely stored on the master node, but in order to distribute the computation across the cluster, data must be partitioned and sent to workers. The HDFS will solve this problem. To upload the data, we copy the two folders with the following commands:

```
hdfs dfs -copyFromLocal data /
hdfs dfs -copyFromLocal panel /
```

and we can see the impact of these actions by looking at Hadoop's disk usage (Figure 21).

The interface also provides the possibility of exploring the file system, where the uploaded dataset will be visible (Figure 22).

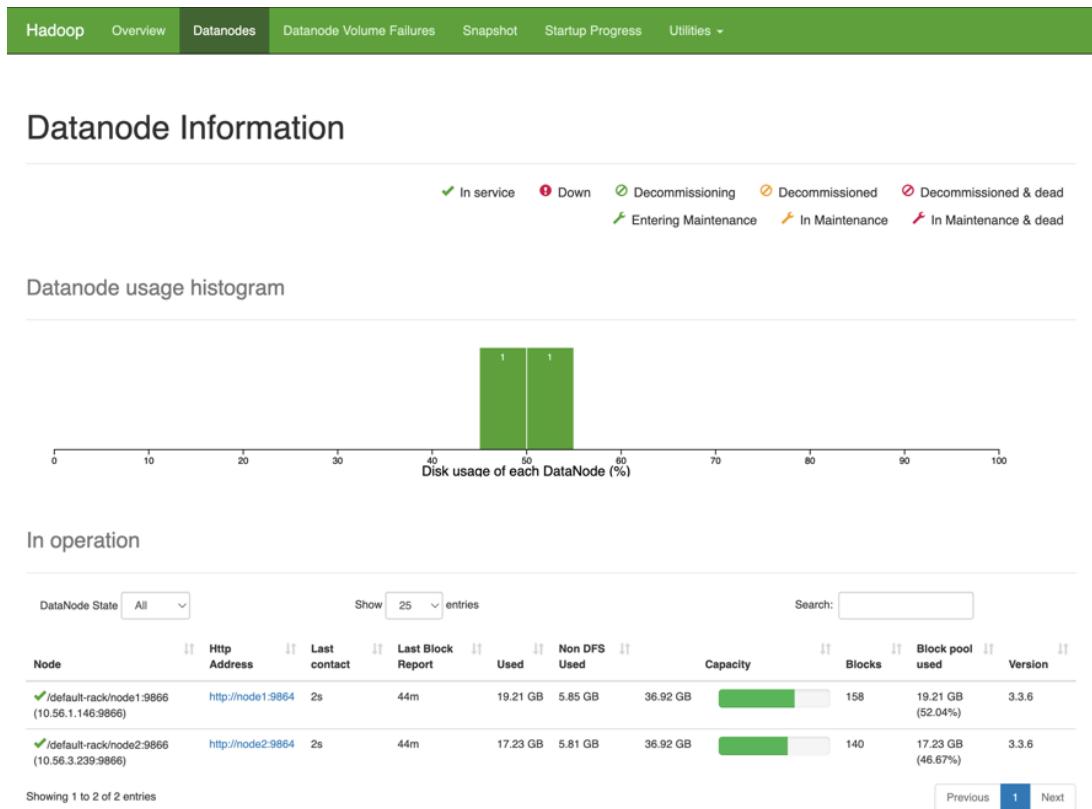


Figure 21

Browse Directory

The screenshot shows the Hadoop Browse Directory page. The URL in the address bar is "/data". The page displays a table of files in the "/data" directory:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	hadoop	supergroup	5.47 GB	Mar 25 02:49	1	128 MB	2018_public_lar_three_year.csv
-rw-r--r--	hadoop	supergroup	6.16 GB	Mar 25 02:44	1	128 MB	2019_public_lar_three_year_csv.csv
-rw-r--r--	hadoop	supergroup	9.33 GB	Mar 25 02:39	1	128 MB	2020_public_lar_one_year.csv
-rw-r--r--	hadoop	supergroup	9.51 GB	Mar 25 02:42	1	128 MB	2021_public_lar_one_year_csv.csv
-rw-r--r--	hadoop	supergroup	5.64 GB	Mar 25 02:46	1	128 MB	2022_public_lar_csv.csv

At the bottom, it says "Showing 1 to 5 of 5 entries".

Figure 22

The Jupyter notebook

- 1) If not already installed we add *pip*, Python's package manager, and we install the pyspark library and jupyter notebook:

```
sudo apt install python3-pip  
pip3 install pyspark  
pip3 install notebook
```

- 2) Once launched with `python3 -m notebook` the notebook can be accessed through port 8888, while the Spark application can be monitored at 4040.
- 3) The notebook, available at [rogerferrod/CCC-2021 \(github.com\)](https://rogerferrod/CCC-2021), will perform the analysis by connecting to the cluster and processing the data. More in detail:

The first block creates a Spark connection with the cluster

```
from pyspark.sql import SparkSession  
from pyspark.sql.functions import concat  
from pyspark.sql.types import StringType  
from pyspark.sql.functions import udf, col  
  
spark = SparkSession.builder \  
    .appName("HDMA analysis") \  
    .master("spark://master:7077") \  
    .config("spark.executor.cores", "1") \  
    .config("spark.executor.memory", "2g") \  
    .getOrCreate()
```

Once created the application will be visible in the web UI at 8080 (Figure 23) and can be monitored at 4040 (Figure 24)

The screenshot shows the Apache Spark web UI for version 3.4.2. At the top, it displays the URL as `spark://master:7077`. Below this, it provides system statistics: URL, Alive Workers, Cores in use, Memory in use, Resources in use, Applications (1 Running, 0 Completed), Drivers (0 Running, 0 Completed), and Status (ALIVE). The main interface is divided into sections: **Workers (2)** and **Running Applications (1)**.

Workers (2)

Worker Id	Address	State	Cores	Memory	Resources
worker-20240325013536-10.56.1.146-46333	10.56.1.146:46333	ALIVE	2 (1 Used)	2.8 GiB (2.0 GiB Used)	
worker-20240325013536-10.56.3.239-46373	10.56.3.239:46373	ALIVE	2 (1 Used)	2.8 GiB (2.0 GiB Used)	

Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20240325015250-0000	(kill) HDMA analysis	2	2.0 GiB		2024/03/25 01:52:50	hadoop	RUNNING	40 s

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Figure 23

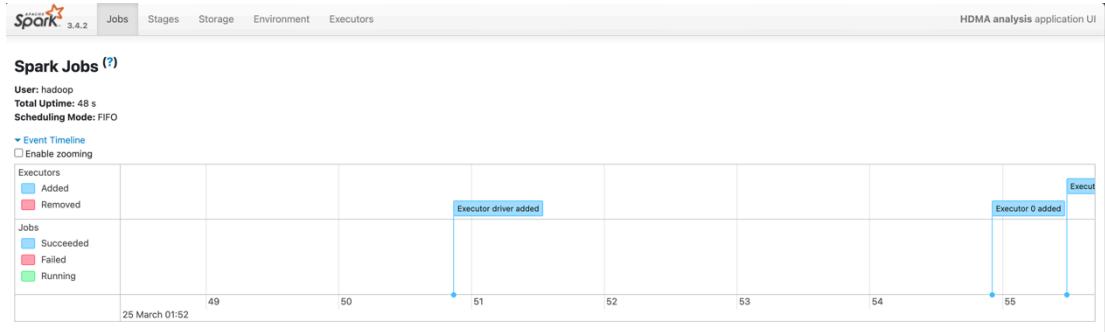


Figure 24

Then we load the data from HDFS

```

panels = ['panel/2018_public_panel_three_year_csv.csv', 'panel/2019_public_panel_three_year_csv.csv',
          | 'panel/2020_public_panel_one_year_csv.csv', 'panel/2021_public_panel_one_year_csv.csv', 'panel/2022_public_panel_csv.csv']

df_panel = None
for ph in panels:
    path = 'hdfs://master:9000/' + ph
    df_panel_sub = spark.read.option("delimiter", ",").option("header", True).option("inferSchema", "true").csv(path)
    lei = 'lei' in df_panel_sub.schema[1].simpleString()
    df_panel_sub = df_panel_sub.withColumn('lei_code', df_panel_sub.lei if lei else df_panel_sub.upper())
    df_panel_sub = df_panel_sub.withColumn('bank', df_panel_sub.other_lender_code == 0)
    df_panel_sub = df_panel_sub.select('lei_code','bank')
    df_panel = df_panel.union(df_panel_sub) if df_panel is not None else df_panel_sub

df_panel = df_panel.dropDuplicates(['lei_code'])

# df = df.withColumn('discount_points', df.discount_points.cast('double'))
# df = df.withColumn('origination_charges', df.origination_charges.cast('double'))
# df = df.withColumn('sum_points', df.origination_charges + df.discount_points)
df = df.withColumn('combined_loan_to_value_ratio', df.combined_loan_to_value_ratio.cast('double'))
df = df.withColumn('total_loan_costs', df.total_loan_costs.cast('double'))

```

and perform a first operation aimed at counting the number of records available in the dataset

```

df.count()

100676029

```

Following the original Bloomberg's analysis, we exclude unlikely records that exceed reasonable parameters or miss important information and we keep only loans for primary residential properties.

```
filtered = df.na.drop(subset=["total_loan_costs", 'interest_rate']).filter(
    (df.derived_loan_product_type.contains('First Lien')) &
    (df.conforming_loan_limit == 'C') &
    (df.reverse_mortgage != 1) &
    (df.interest_only_payment != 1) &
    (df.construction_method != 2) &
    (df.loan_purpose == 1) &
    (df.occupancy_type == 1) &
    (df.total_loan_costs <= 50000) &
    (df.total_loan_costs <= df.loan_amount) &
    # (df.total_loan_costs <= df.sum_points) &
    (df.combined_loan_to_value_ratio <= 102)
)

filtered.count()
```

16762105

Then, with a user-defined function applied to each record we obtain the ethnicity of applicants and store the information in a new column.

```
@udf
def get_race(race, ethnicity):
    for r in ['White', 'Black', 'Asian']:
        if r in race:
            return r

    if ethnicity == 'Hispanic or Latino':
        return 'Latino'

    return None

filtered = filtered.withColumn("ethnicity", get_race(col('derived_race'), col('derived_ethnicity')))
```

Since the LAR table does not provide institutional information and other details about the lender provider, a join with the Reporter Panel is needed. After the operation, only a selection of columns will be kept.

```
joined = filtered.join(df_panel, filtered.lei == df_panel.lei_code)
joined = joined.select('bank','ethnicity','activity_year','total_loan_costs','income',
                      'debt_to_income_ratio','loan_amount','combined_loan_to_value_ratio','discount_points',
                      'lender_credits','loan_type','loan_purpose')

joined = joined.withColumn('income', df.income.cast('double'))
joined = joined.withColumn('debt_to_income_ratio', df.debt_to_income_ratio.cast('double'))
joined = joined.withColumn('lender_credits', df.lender_credits.cast('double'))

joined = joined.na.drop('any')
```

We can now have a glimpse of the differences between bank and non-bank institutions by looking and the average loan cost for each year.

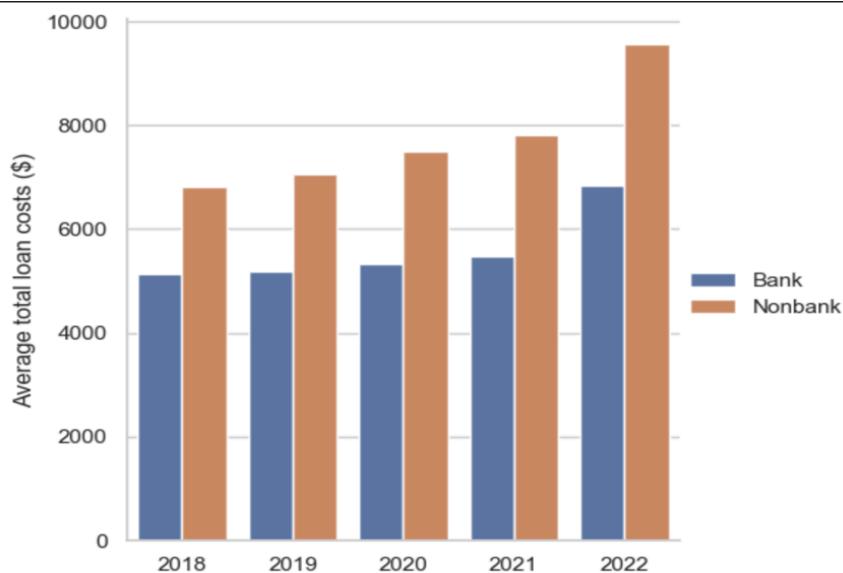
```
per_year_avg = joined.groupBy('activity_year','bank').avg('total_loan_costs').toPandas()

import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="whitegrid")

per_year_avg['bank'] = per_year_avg['bank'].apply(lambda x:'Bank' if x == True else 'Nonbank')

g = sns.catplot(
    data=per_year_avg, kind="bar",
    x="activity_year", y="avg(total_loan_costs)", hue="bank"
)

g.set_axis_labels("", "Average total loan costs ($)")
g.legend.set_title("")
```



Not surprisingly non-bank lenders ask for higher fees for the service, but to further investigate the discrepancy between ethnic groups and lender types we first need to separate bank and non-bank loans and perform a few linear regression analyses.

```
bank = joined.filter(joined.bank == True)
nonbank = joined.filter(joined.bank != True)
```

The multiple linear regression analysis is meant to study the difference in total loan cost between borrowers of comparable characteristics, such as income, interest rates, loan size, debt-to-income ratio, etc. To this aim, we train a linear regression model on white borrowers (both from bank and non-bank lenders) asking the model to predict the total loan costs (dependent variable) given a set of independent variables that do not include race.

```
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorAssembler

assembler = VectorAssembler(inputCols=['income','debt_to_income_ratio','loan_amount',
                                       'combined_loan_to_value_ratio','discount_points',
                                       'lender_credits','loan_type','loan_purpose'],
                            outputCol='features')

lr = LinearRegression(featuresCol='features', labelCol='total_loan_costs', predictionCol='pred_cost', regParam=0.5, elasticNetParam=0.8)

white_data_bank = assembler.transform(
    bank.filter(bank.ethnicity.contains('White'))
).select('features', 'total_loan_costs')

white_data_nonbank = assembler.transform(
    nonbank.filter(nonbank.ethnicity.contains('White'))
).select('features', 'total_loan_costs')

lr_bank = lr.fit(white_data_bank)
lr_nonbank = lr.fit(white_data_nonbank)
```

Once the models are trained, we can predict which cost should face non-white borrowers with a comparable fiscal life and confront the difference with the actual costs asked by the loan providers.

```
data_bank = assembler.transform(bank.filter(bank.ethnicity != 'White')).select('features', 'total_loan_costs', 'ethnicity')
data_nonbank = assembler.transform(nonbank.filter(bank.ethnicity != 'White')).select('features', 'total_loan_costs', 'ethnicity')

nonbank_pred = lr_nonbank.transform(data_nonbank)
bank_pred = lr_bank.transform(data_bank)

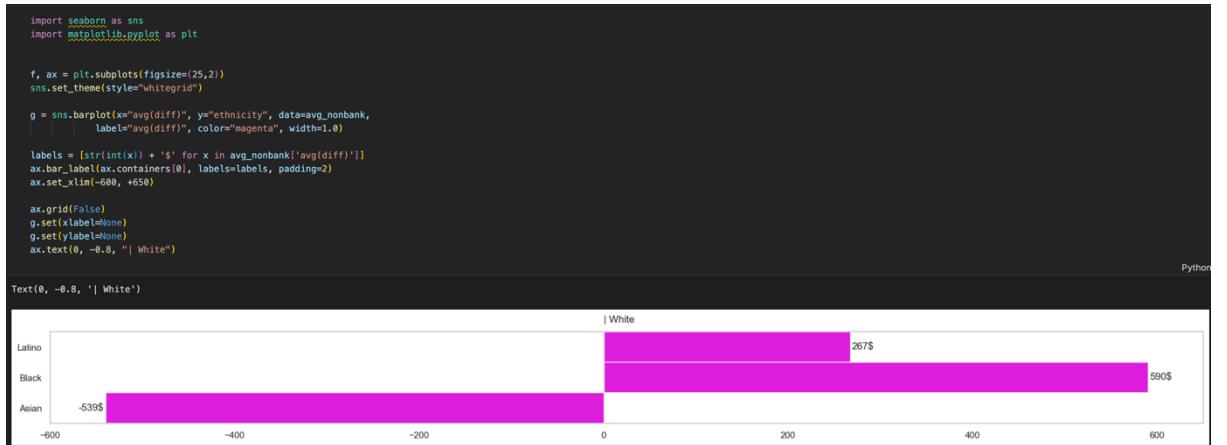
nonbank_pred = nonbank_pred.withColumn('diff', nonbank_pred.total_loan_costs - nonbank_pred.pred_cost)
bank_pred = bank_pred.withColumn('diff', bank_pred.total_loan_costs - bank_pred.pred_cost)
```

We collect the data in two pandas dataframes

```
from pyspark.sql.functions import mean

avg_nonbank = nonbank_pred.groupBy('ethnicity').avg('diff').toPandas()
avg_bank = bank_pred.groupBy('ethnicity').avg('diff').toPandas()
```

and plot the results



The two plots show the difference in total loan costs compared to white borrowers for bank institutions (black plot) and non-bank lenders (magenta).

While Asians enjoy lower fees, on average, if compared to other ethnicities, Blacks and Latinos pay a higher price with respect to White borrowers with similar characteristics. Differences are exacerbated if the loads are provided by non-bank lenders.

- 4) Throughout the execution of the analysis, jobs can be monitored with the user interface provided by Spark, where details about the computing graph (DAG) and execution timeline are shown (Figure 25,26,27,28)

The screenshot shows the 'Spark Jobs' interface. At the top, it displays basic information: User: hadoop, Total Uptime: 13 min, Scheduling Mode: FIFO, Active Jobs: 1, and Completed Jobs: 14. Below this is the 'Event Timeline' section, which is currently collapsed. The main area is divided into two tabs: 'Active Jobs (1)' and 'Completed Jobs (14)'. The 'Active Jobs' tab shows one job (Job ID 14) with a description of 'collect at /tmp/pykernel_33278/159842356.py:1' and 'collect at /tmp/pykernel_33278/159842356.py:1 (kill)'. The 'Submitted' time is 2024/03/25 16:33:47, Duration is 21 s, and Stages: Succeeded/Total is 0/1. The 'Tasks (for all stages): Succeeded/Total' is 10/290 (2 running). The 'Completed Jobs' tab lists 14 completed jobs, each with a description, Submitted time, Duration, Stages: Succeeded/Total, and Tasks (for all stages): Succeeded/Total. For example, Job 13 (dca3a261-f50e-4c44-ac5f-de72d241fc2c) has a duration of 0.6 s, 1/1 (1 skipped) stages, and 1/1 tasks. Job 12 has a duration of 2 s, 1/1 stages, and 5/5 tasks. Job 11 has a duration of 11 min, 1/1 stages, and 290/290 tasks. Job 10 has a duration of 0.2 s, 1/1 stages, and 1/1 tasks. Job 9 has a duration of 2 s, 1/1 stages, and 1/1 tasks. Job 8 has a duration of 0.2 s, 1/1 stages, and 1/1 tasks.

Figure 25

The screenshot shows the 'Executors' interface. It starts with a 'Summary' table that provides a high-level overview of executor resources. The table includes columns for Active(3), Dead(0), and Total(3) executors, detailing their memory usage (1.1 MiB / 2.5 GiB), disk usage (0.0 B), and core counts (2 cores). Below this is a detailed 'Executors' table listing individual executors. The table has columns for Executor ID, Address, Status, RDD Blocks, Storage Memory, Disk Used, Cores, Active Tasks, Failed Tasks, Complete Tasks, Total Tasks, Task Time (GC Time), Input, Shuffle Read, Shuffle Write, and Excluded. Three executors are listed: Executor 0 (Address: 10.56.3.239:44711, Active, 0 RDD blocks, 321.9 KiB / 1 GiB storage, 0.0 B disk used, 1 core, 2 active tasks, 0 failed tasks, 181 complete tasks, 183 total tasks, 12 min (9 s) task time, 0.0 B input, 249.9 KiB shuffle read, 249.9 KiB shuffle write, 0 logs, and Thread Dump), Executor driver (Address: master:39247, Active, 0 RDD blocks, 426.2 KiB / 434.4 MiB storage, 0.0 B disk used, 0 cores, 0 active tasks, 0 failed tasks, 0 complete tasks, 0 total tasks, 14 min (0.6 s) task time, 0.0 B input, 0.0 B shuffle read, 0.0 B shuffle write, 0 logs, and Thread Dump), and Executor 1 (Address: 10.56.1.146:35829, Active, 0 RDD blocks, 356.7 KiB / 1 GiB storage, 0.0 B disk used, 1 core, 2 active tasks, 0 failed tasks, 170 complete tasks, 172 total tasks, 12 min (9 s) task time, 19.6 GiB input, 601.4 KiB shuffle read, 357.7 KiB shuffle write, 0 logs, and Thread Dump). A search bar and navigation buttons (Previous, Next) are also present.

Figure 26

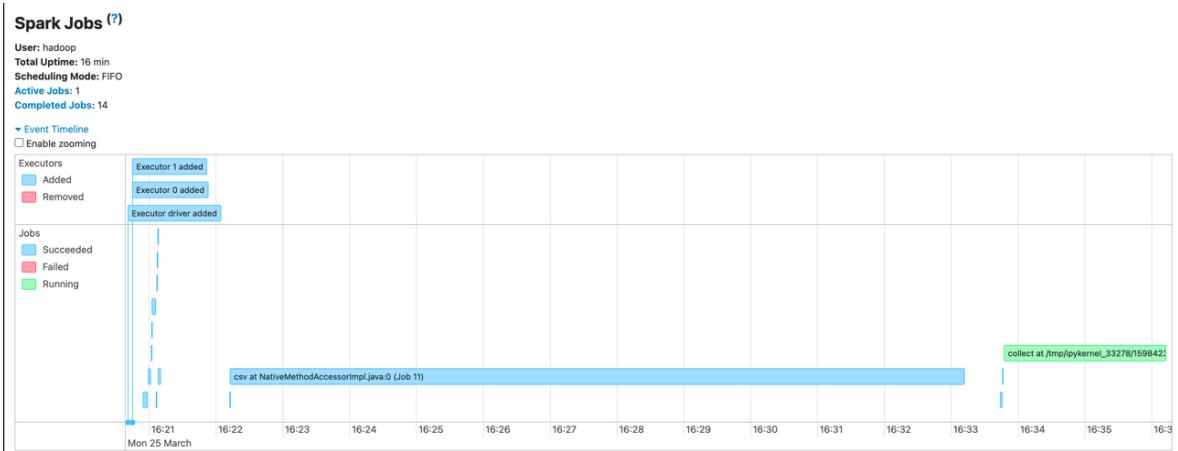


Figure 27

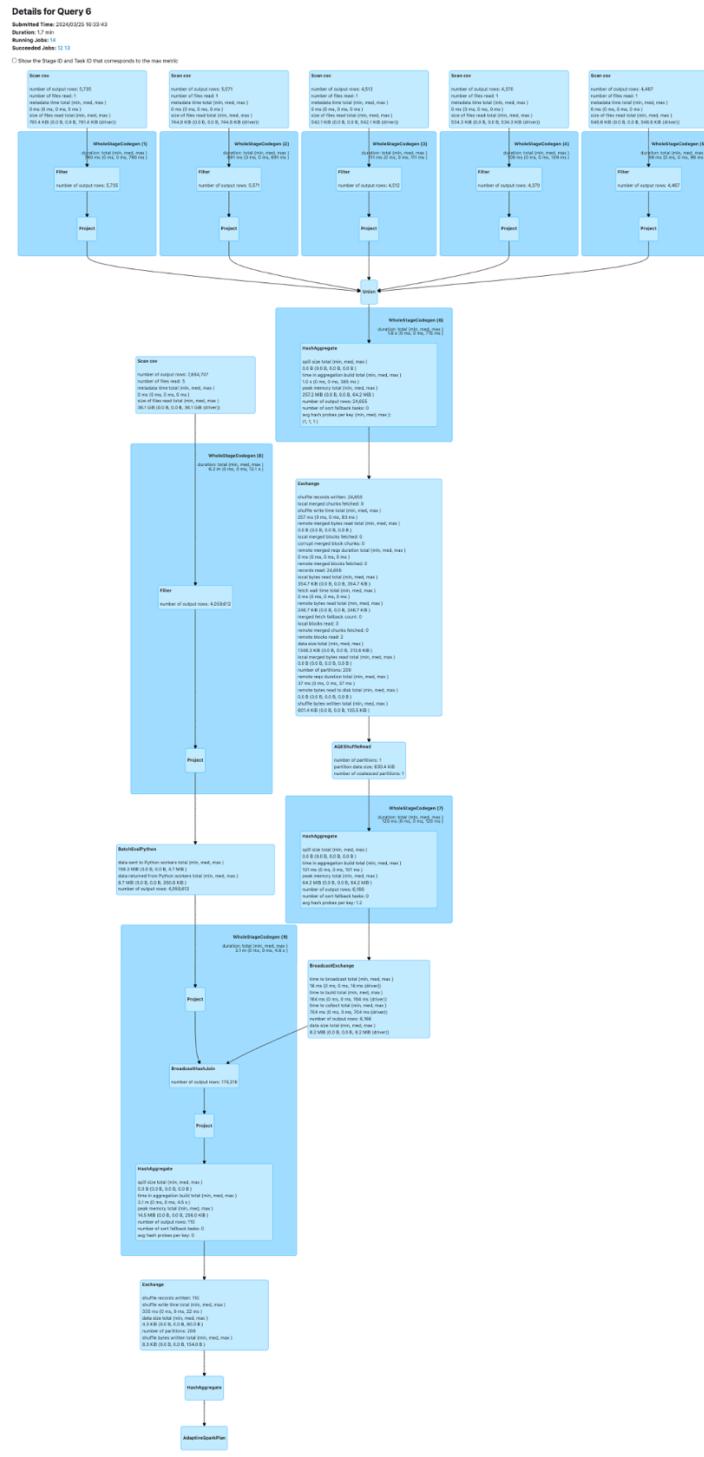


Figure 28

5) Spark, YARN and Hadoop services can be stopped, respectively, with:

```
$SPARK_HOME/sbin/stop-all.sh
stop-yarn.sh
stop-dfs.sh
```