Analysis on mortgages cost with an Apache Spark cluster

Roger Ferrod, University of Turin

The aim of this tutorial is to replicate a Bloomberg analysis[1] 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 database[2], a comprehensive collection of the mortgage maker activities that lender providers report to their regulators. For the purpose of this study, some multiple linear regression analyses are needed, as well as a preprocessing phase aim 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 is Figure 1.

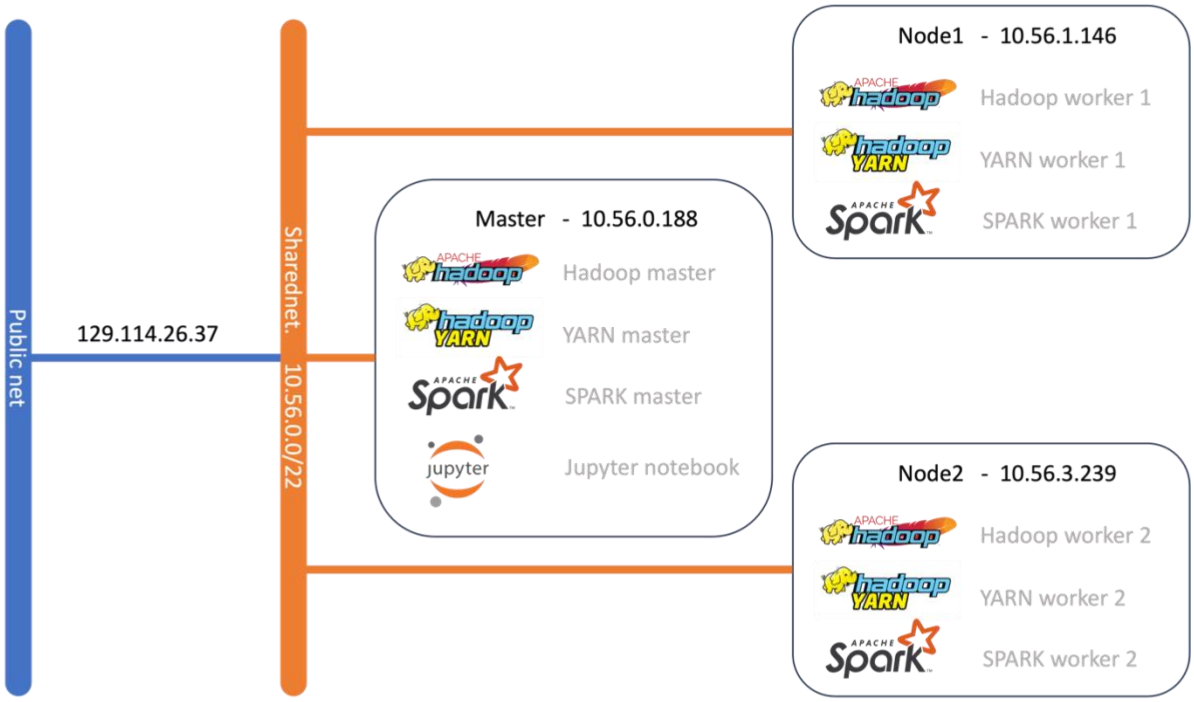


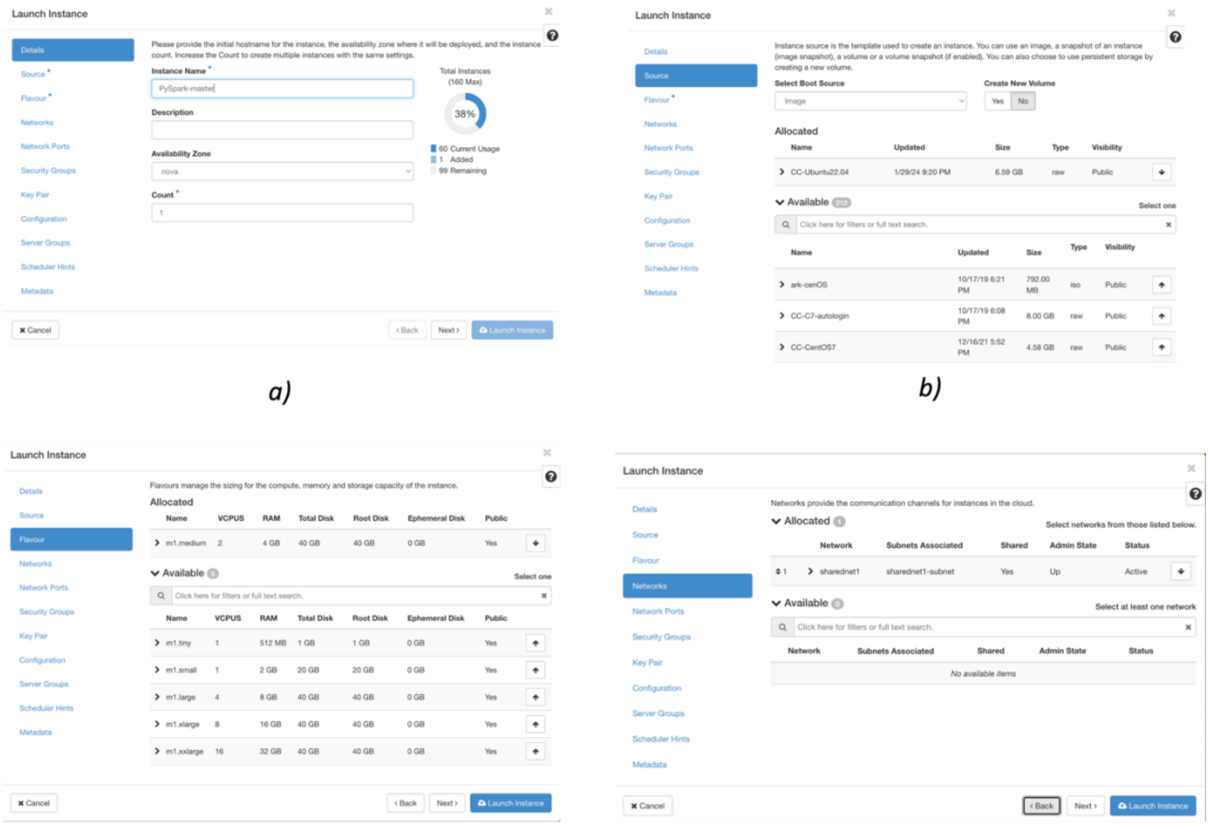
Figure 1

---

[1] https://www.bloomberg.com/graphics/2023-nonbank-lender-mortgage-loan-borrower-fee/
[2] https://ffiec.cfpb.gov/

Create and initialize the virtual machines

1) 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

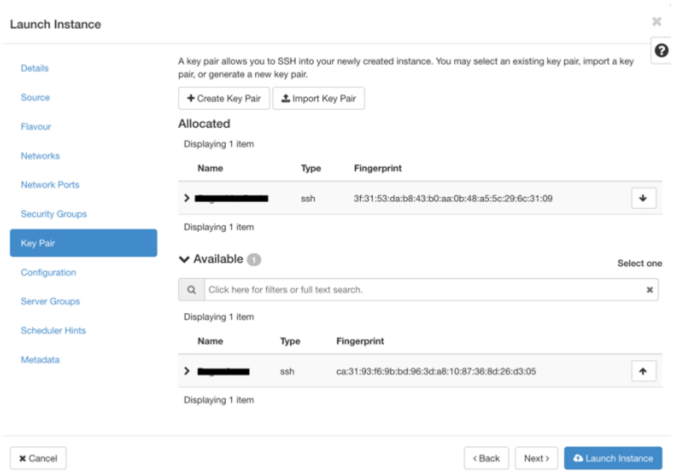


Figure 3

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

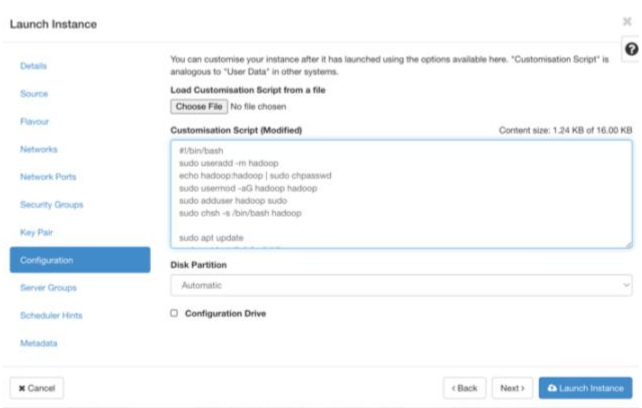
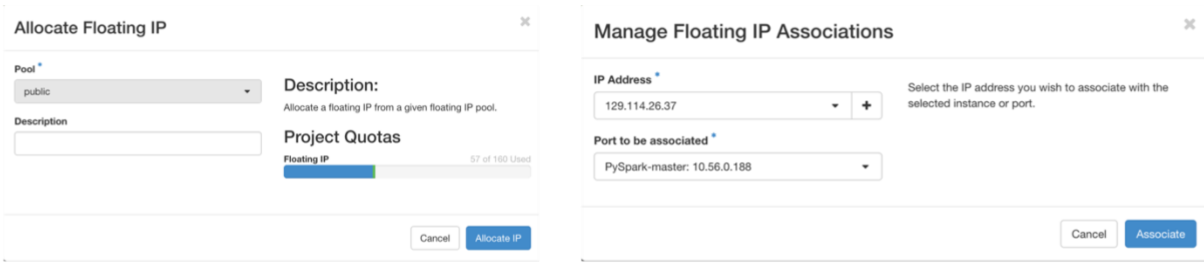


Figure 4

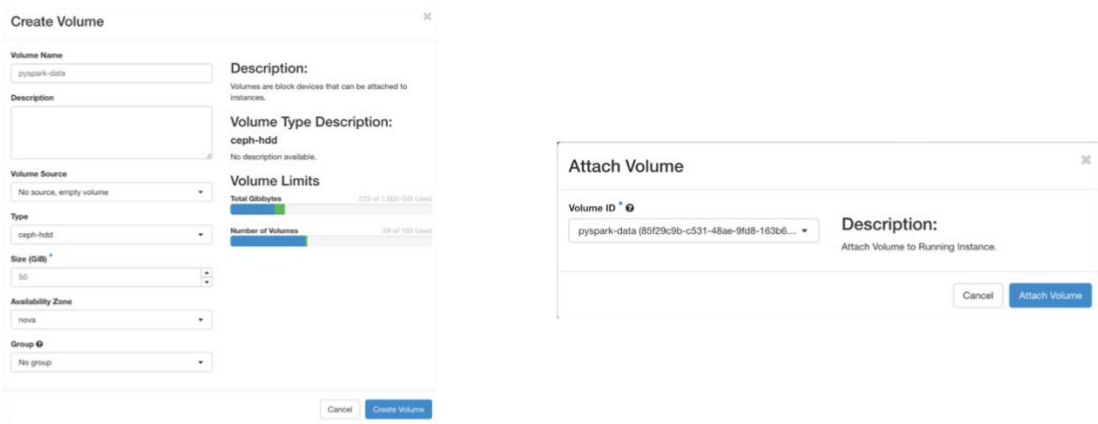
2) Meanwhile we can allocate a public IP (Figure 5 a) and associate it to the instance (Figure 5 b) in order to reach the master node from the external network.



a) *b)*

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).



a) *b)*

Figure 6

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

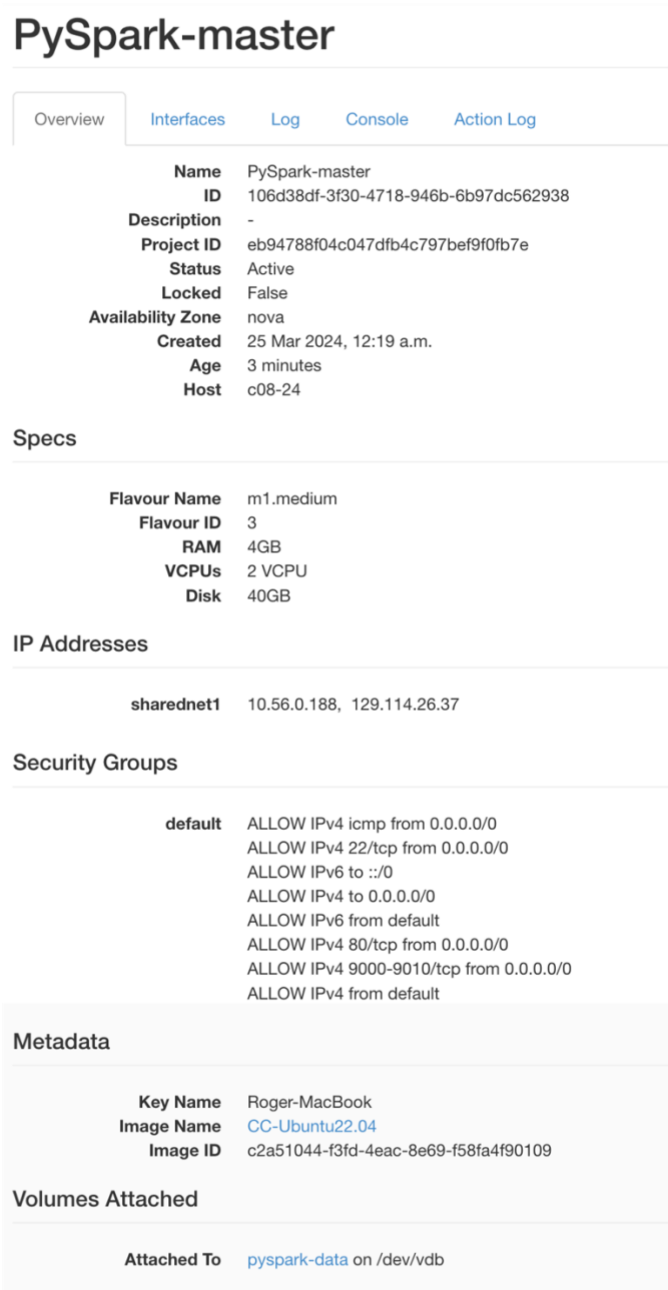


Figure 7

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

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

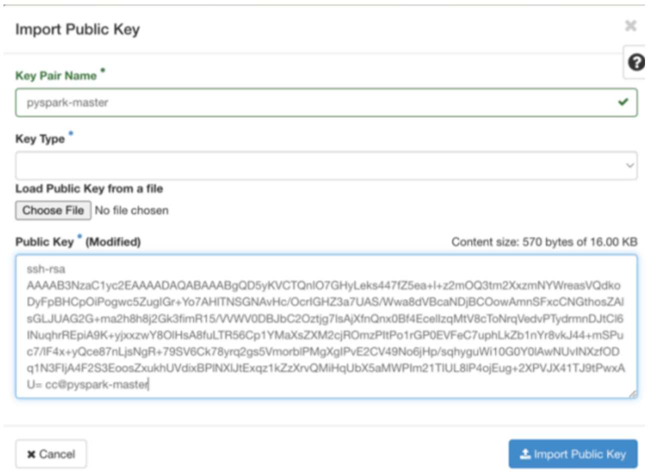


Figure 8

5) At this point we can create and launch two virtual machines that will serve as workers. From the OpenStack web interface we set 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).

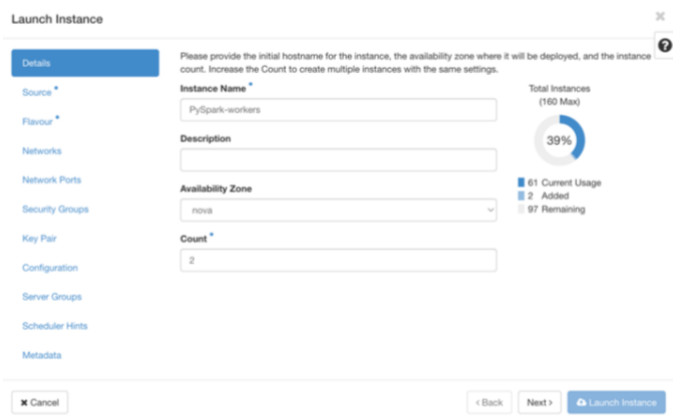


Figure 9

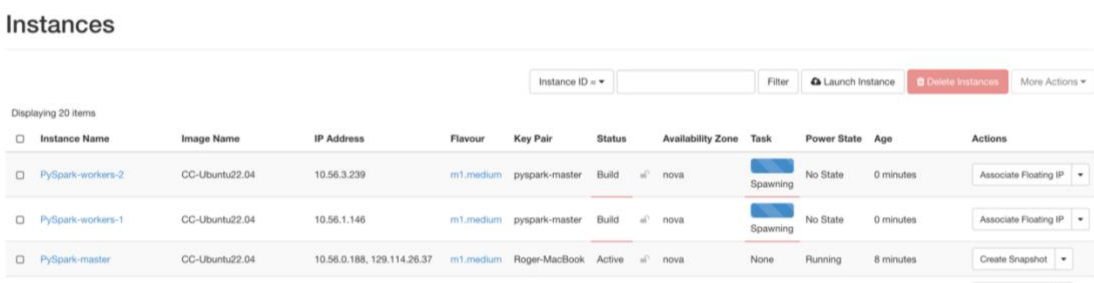


Figure 10

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 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 a *ssh key-pair* for also the hadoop user, we copy the master's public key and paste it on 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
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>
```

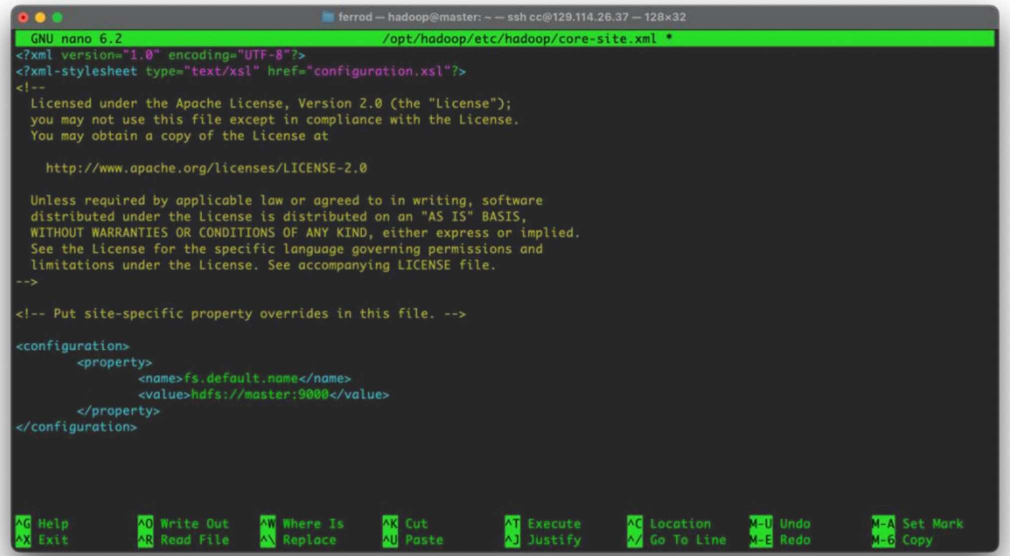


Figure 11

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 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 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).

Figure 12

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-tunel can be open, 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, that in our case should be 2. In the Datanodes section we can monitor the disk usage of worker's nodes (Figure 15).

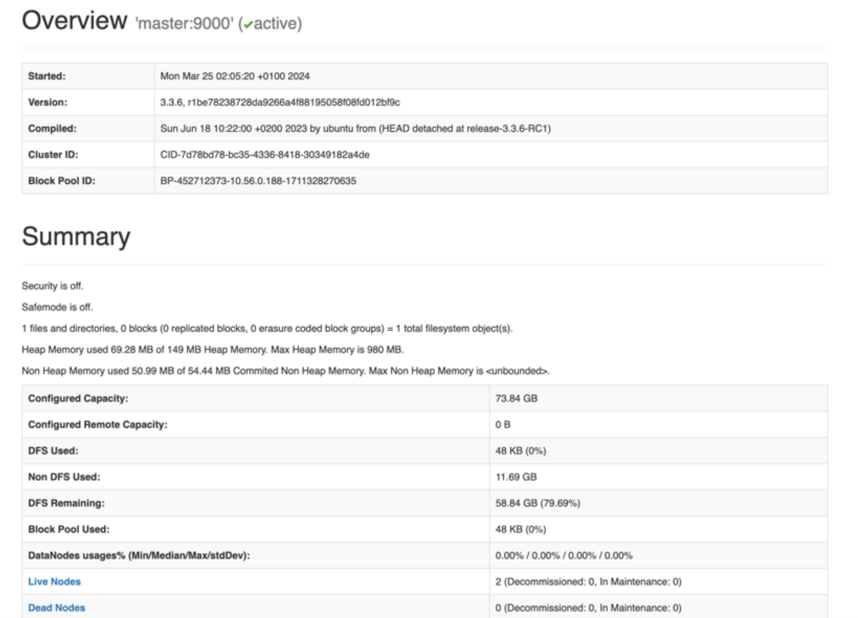


Figure 14

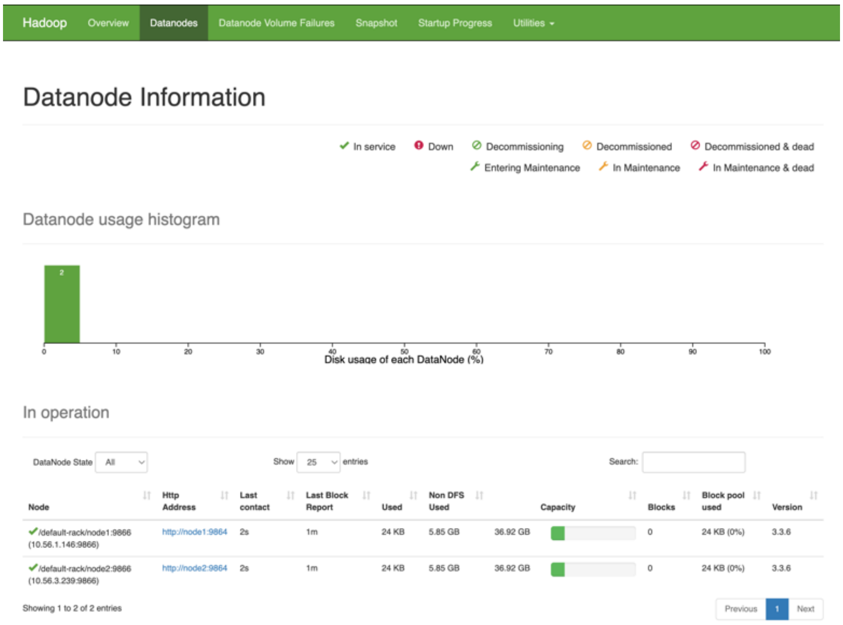


Figure 15

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.

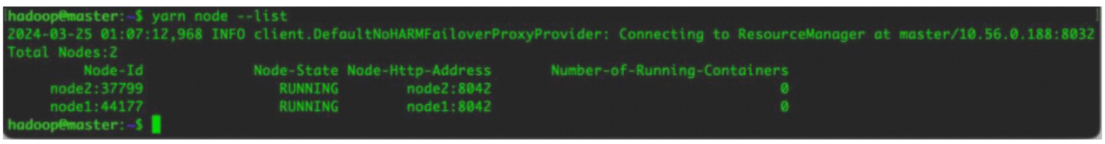


Figure 16

7) Finally, we set up Apache-Spark by editing its configuration files, more in details 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 launch 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.

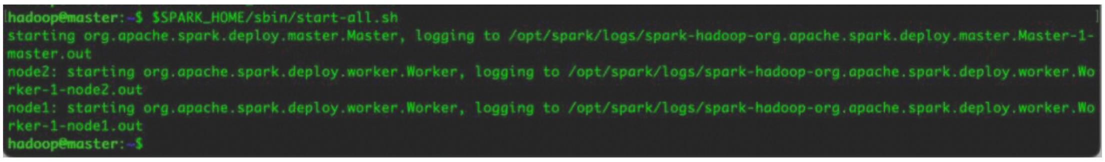


Figure 17

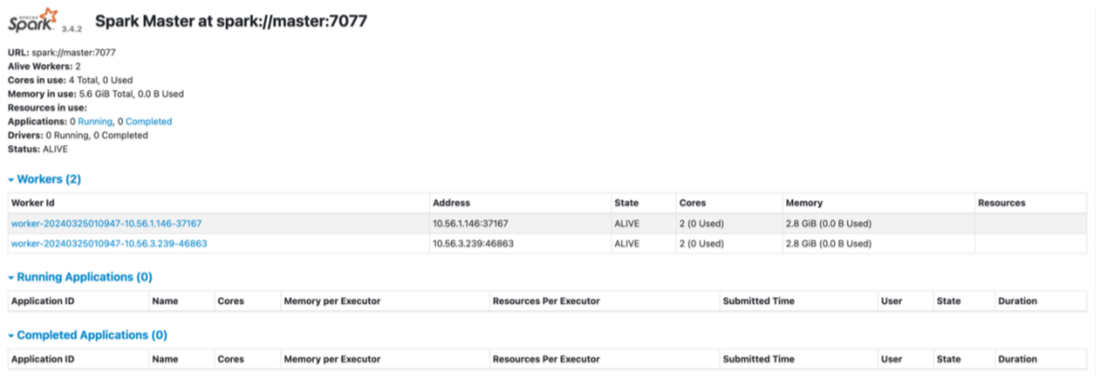


Figure 18

Prepare the dataset

1) Since the 40GB of storage provided by the master node are 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
hadoop@master:~$
```

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 permanent the mount point 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_csv.zip https://s3.amazonaws.com/cfpb-hmda-public/prod/three-year-data/2019/2019_public_lar_three_year_csv.zip https://s3.amazonaws.com/cfpb-hmda-public/prod/one-year-data/2020/2020_public_lar_one_year_csv.zip https://s3.amazonaws.com/cfpb-hmda-public/prod/one-year-data/2021/2021_public_lar_one_year_csv.zip 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_csv.zip https://s3.amazonaws.com/cfpb-hmda-public/prod/one-year-data/2021/2021_public_panel_one_year_csv.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).

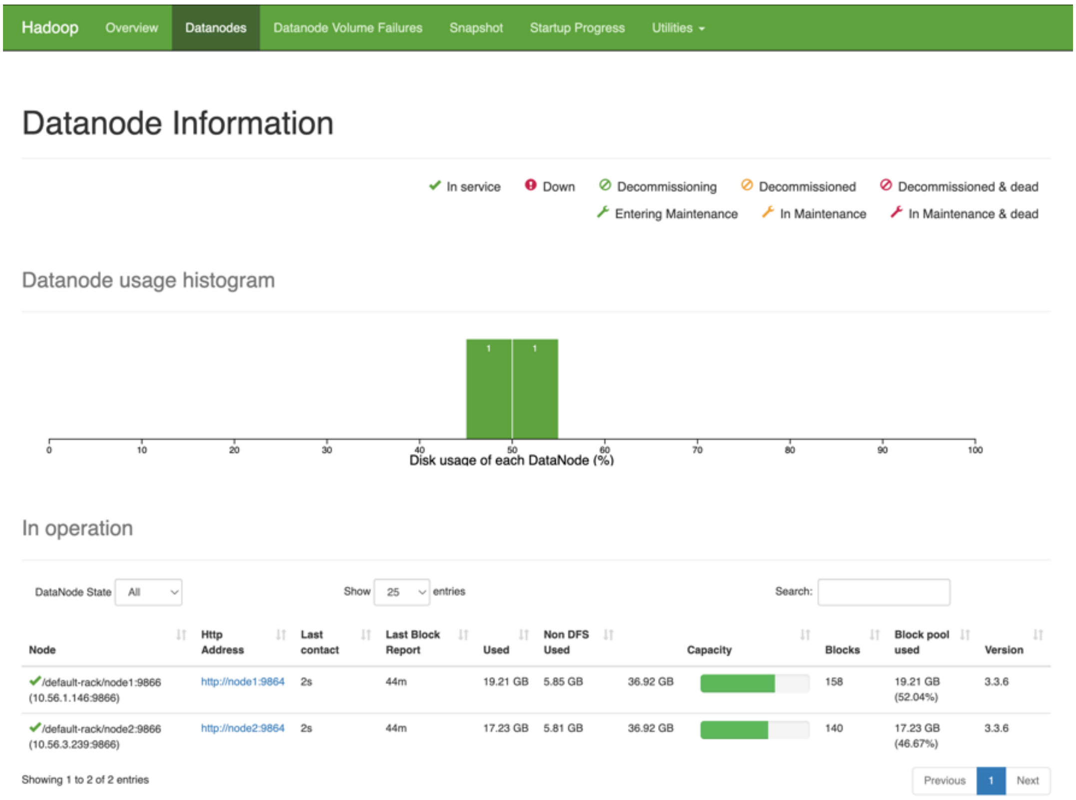


Figure 21

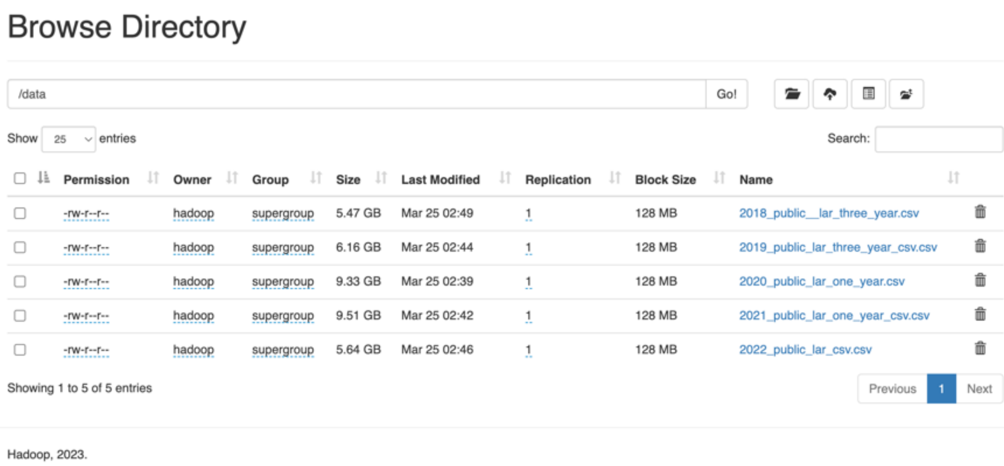


Figure 22

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 launch 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)](), will perform the analysis by connecting to the cluster and processing the data. More in details:

   The first block creates a Spark connection with the cluster

```python
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)

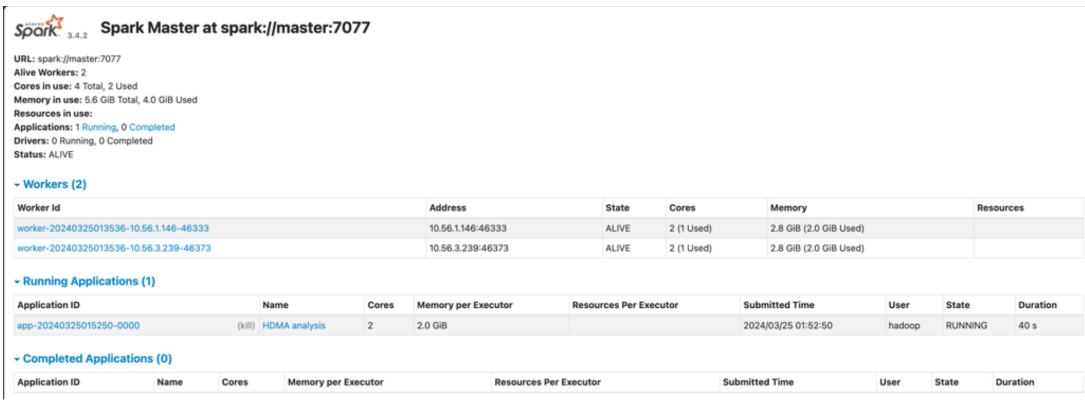


Figure 23

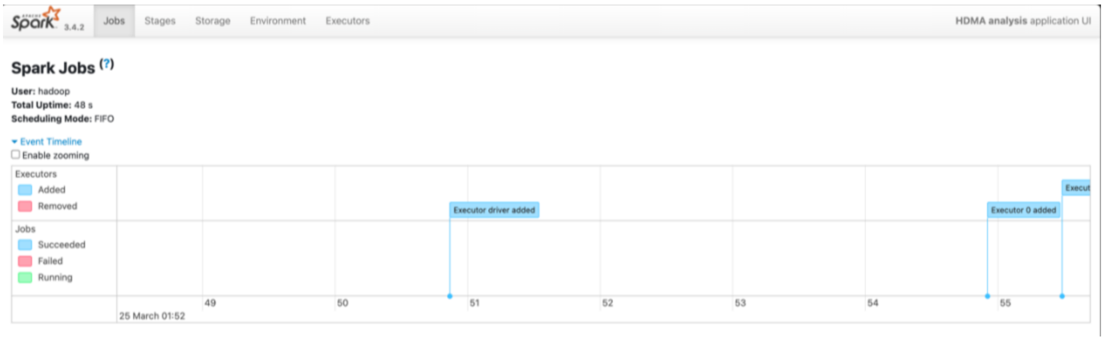


Figure 24

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'])
```

```
path = "hdfs://master:9000/data"
df = spark.read.option("delimiter", ",").option("header", True).option("inferSchema", "true").csv(path)
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
```

Them, 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 on the differences between bank and non-bank institutions by looking and the average loan cost for each year.

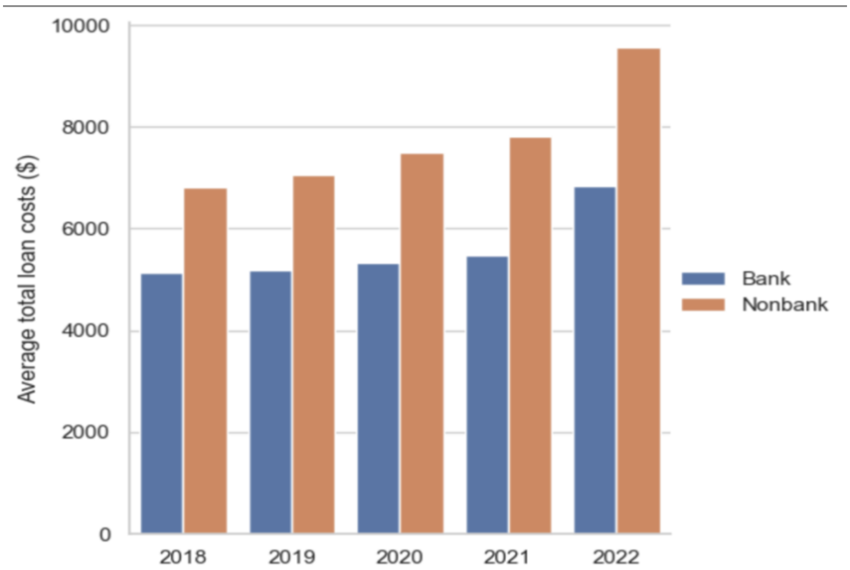
```
per_year_avg = joined.groupBy('activity_year','bank').avg('total_loan_costs').toPandas()
```

```
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 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 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 the 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

```python
import seaborn as sns
import matplotlib.pyplot as plt


f, ax = plt.subplots(figsize=(25,2))
sns.set_theme(style="whitegrid")

g = sns.barplot(x="avg(diff)", y="ethnicity", data=avg_nonbank,
                label="avg(diff)", color="magenta", width=1.0)

labels = [str(int(x)) + '$' for x in avg_nonbank['avg(diff)']]
ax.bar_label(ax.containers[0], labels=labels, padding=2)
ax.set_xlim(-600, +650)

ax.grid(False)
g.set(xlabel=None)
g.set(ylabel=None)
ax.text(0, -0.8, "| White")
```

Python

```
Text(0, -0.8, '| White')
```


```python
f, ax = plt.subplots(figsize=(25,2))
sns.set_theme(style="whitegrid")

g = sns.barplot(x="avg(diff)", y="ethnicity", data=avg_bank,
                label="avg(diff)", color="black", width=1.0)

labels = [str(int(x)) + '$' for x in avg_bank['avg(diff)']]
ax.bar_label(ax.containers[0], labels=labels, padding=2)
ax.set_xlim(-600, +650)

ax.grid(False)
g.set(xlabel=None)
g.set(ylabel=None)
ax.text(0, -0.8, "| White")
```

Python

```
Text(0, -0.8, '| White')
```

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 Asian enjoy lower fees, on average, if compared to other ethnicities, Black 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)

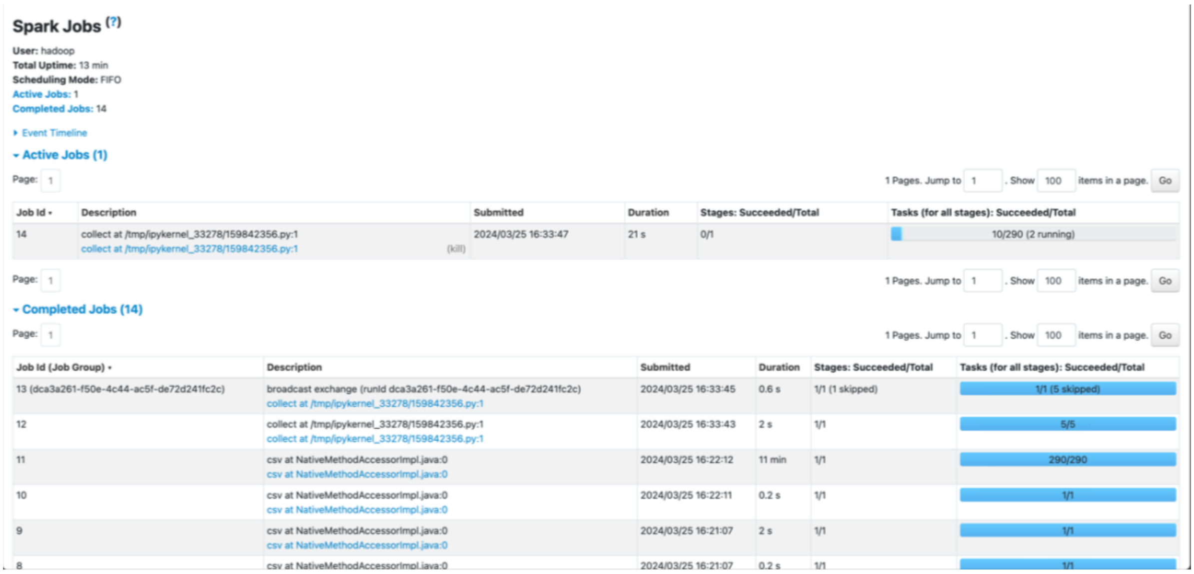


Figure 25

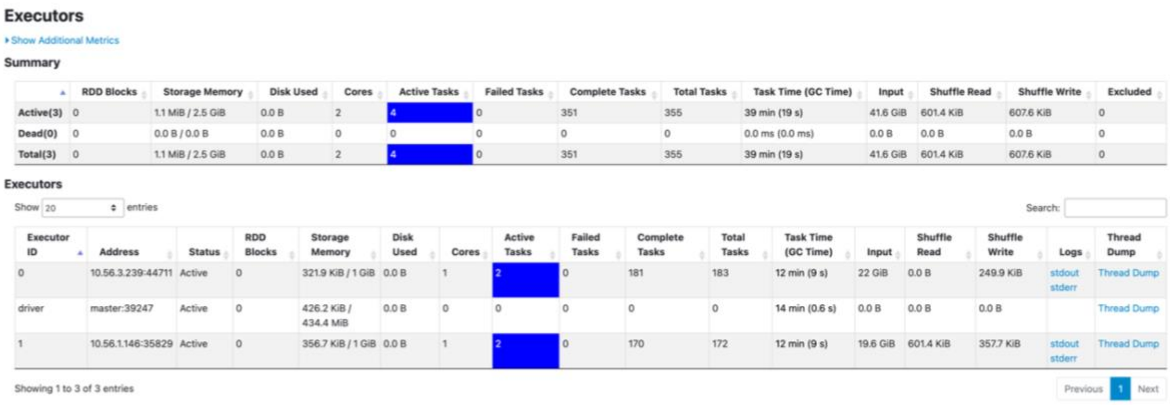


Figure 26

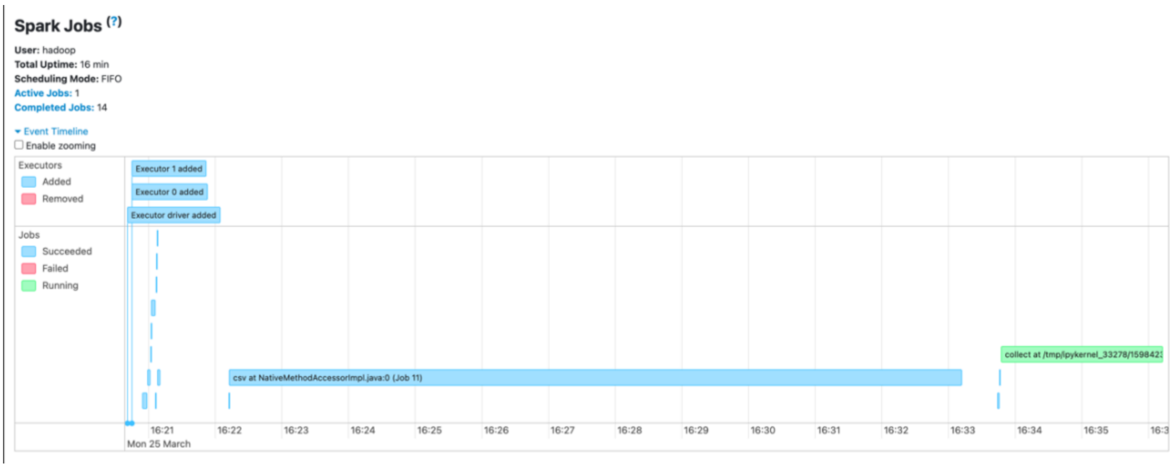


Figure 27

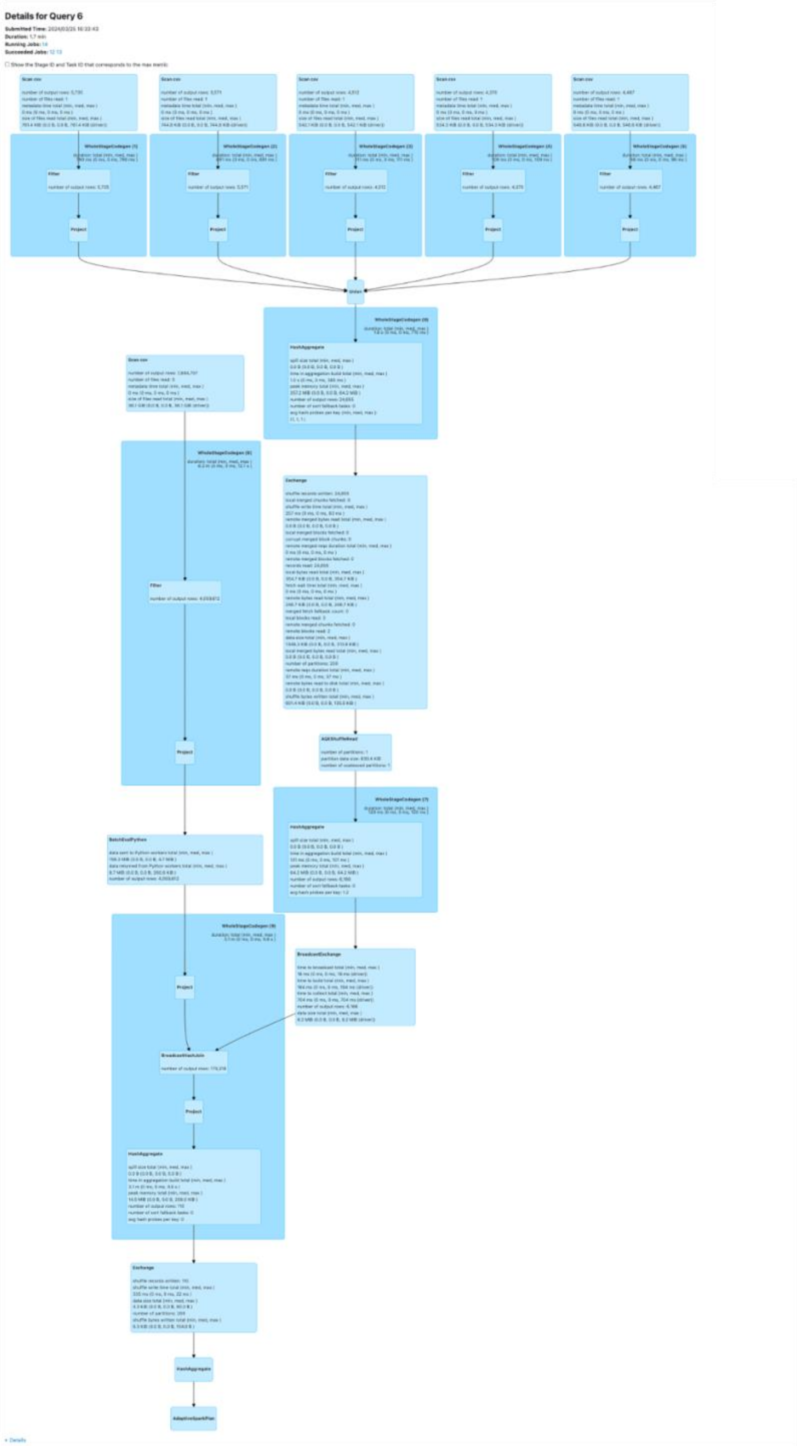


Figure 28

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

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