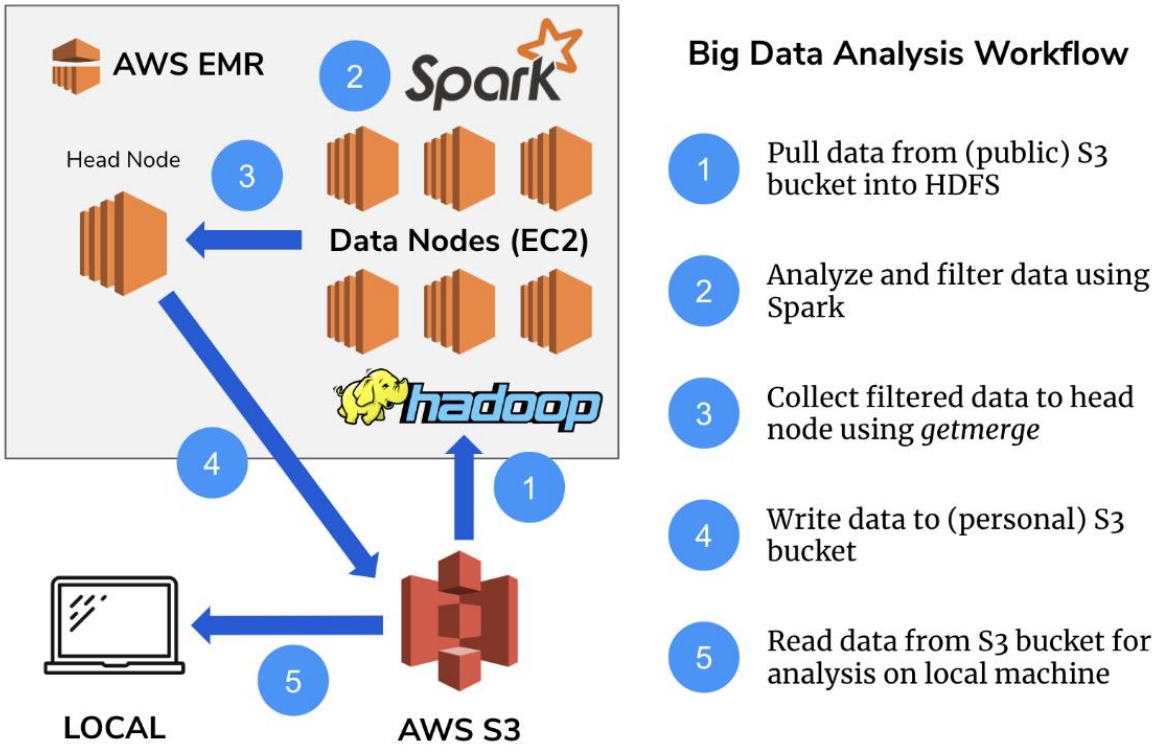
Mahyar Sabouniaghdam **BrainStation**

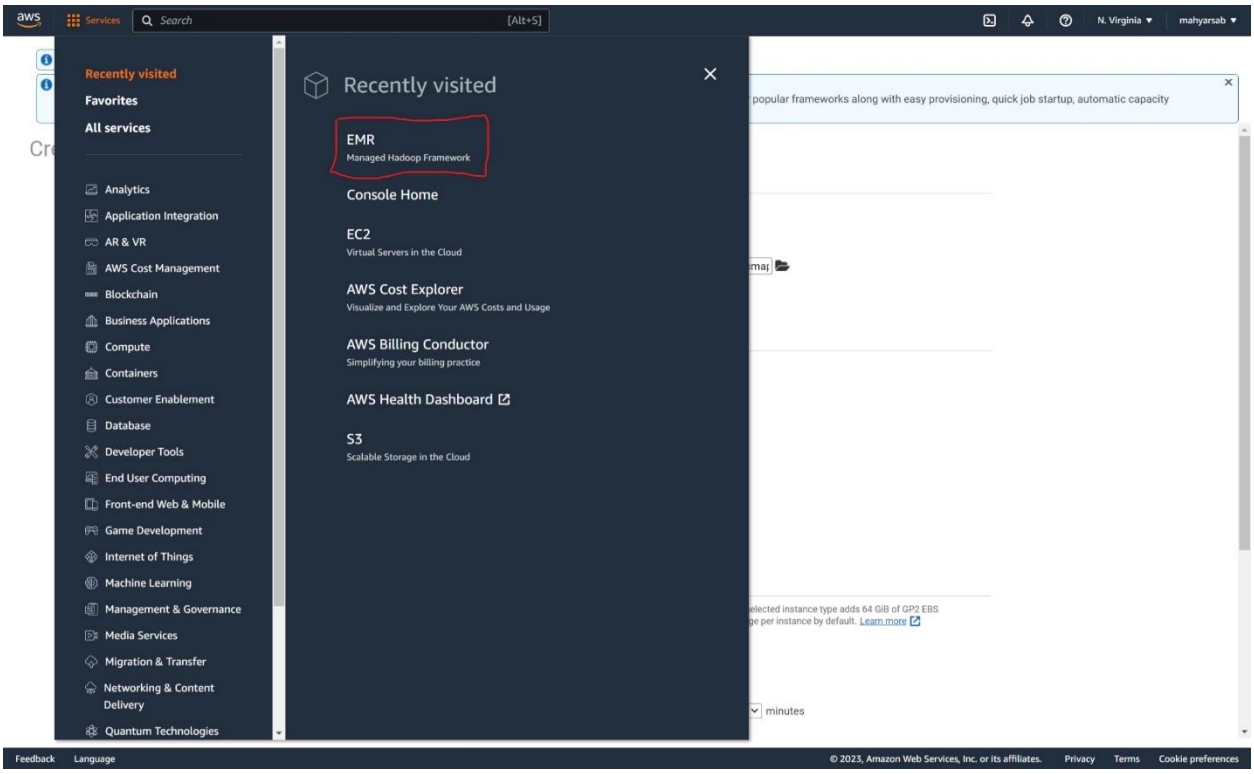
Big Data Wrangling with Google Books Ngrams

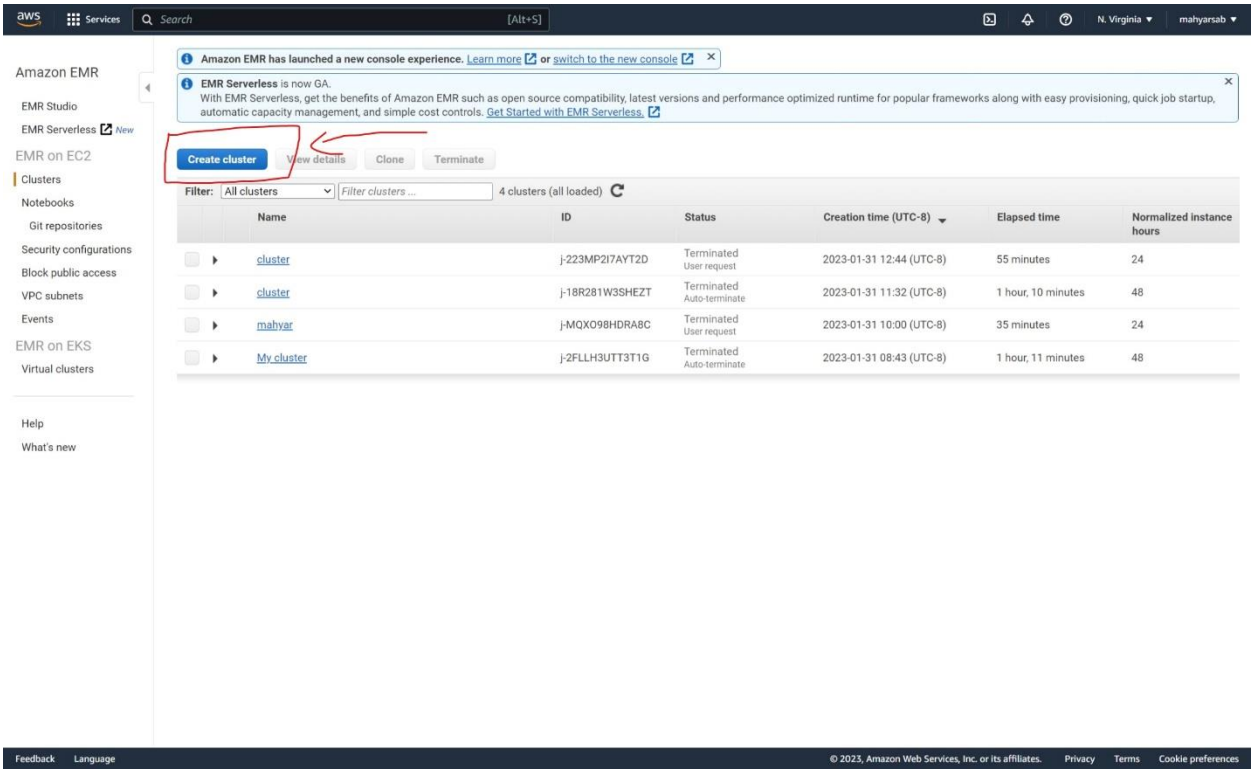
For this deliverable, you will produce a report, as well as a jupyter notebook, which will follow a Big Data analysis workflow. As part of this workflow you will filter and reduce data down to a manageable size, and then do some analysis locally on our machine after extracting data from the Cloud and processing it using Big Data tools. The workflow and steps in the process are illustrated below:



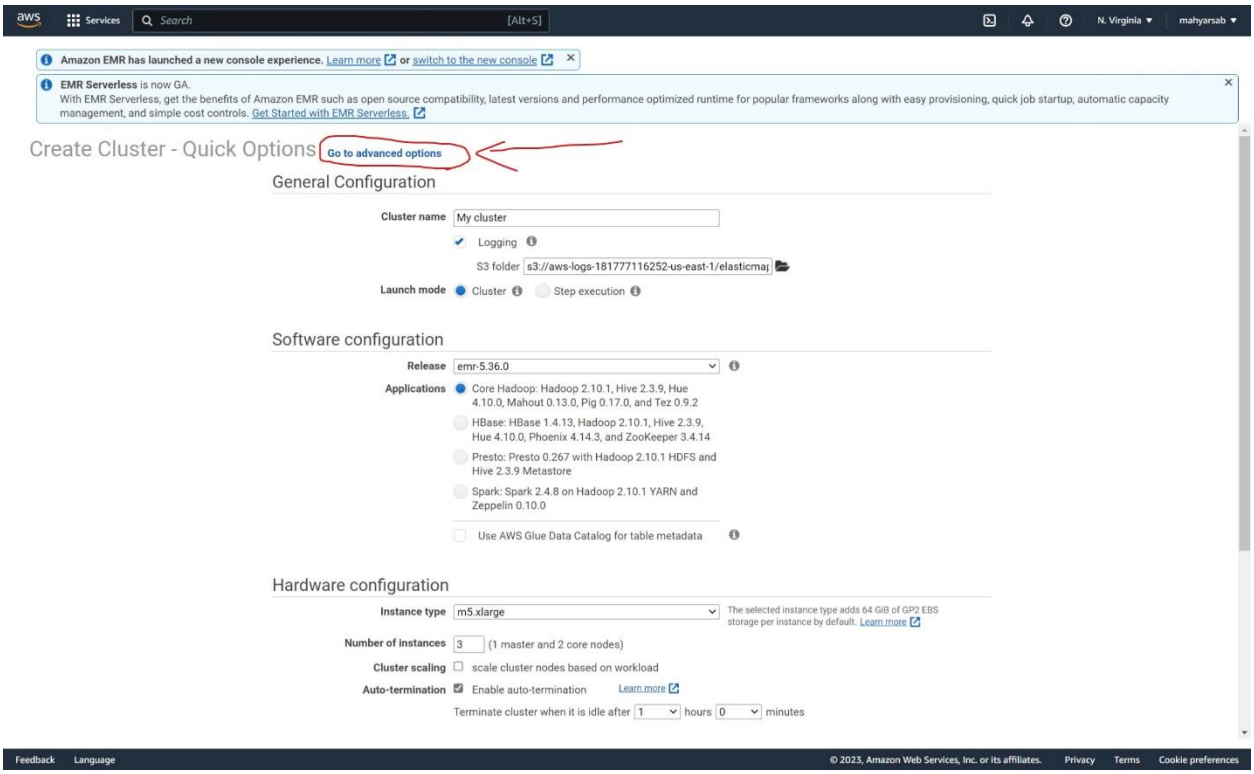
Question 1: Spin up a new EMR cluster using the AWS Console. Be sure to include Hadoop, Spark, Hive, Jupyterhub, and Livy for your cluster. **For the release version, make sure to use EMR 6.1.1:**

1. First we will sign in to our amazon aws account and luanch an EMR cluster like the steps below:

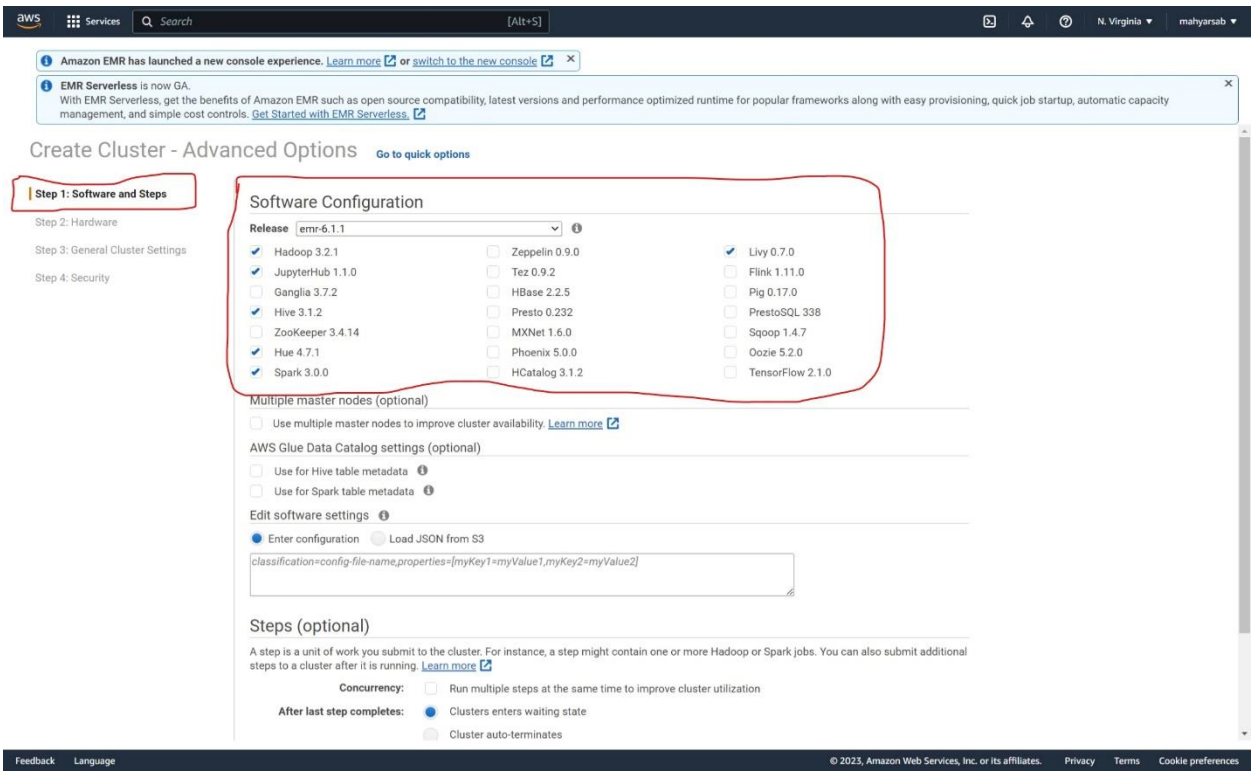




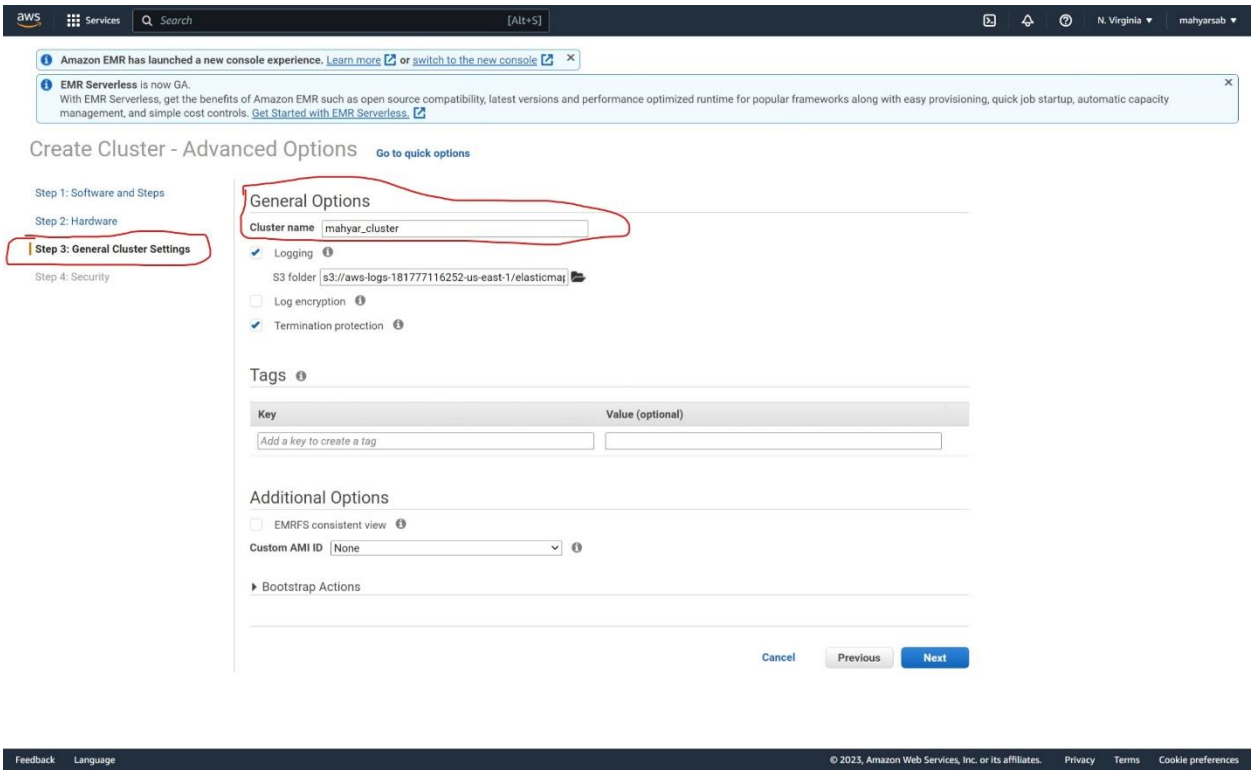
- Click on create cluster and go to advanced options:



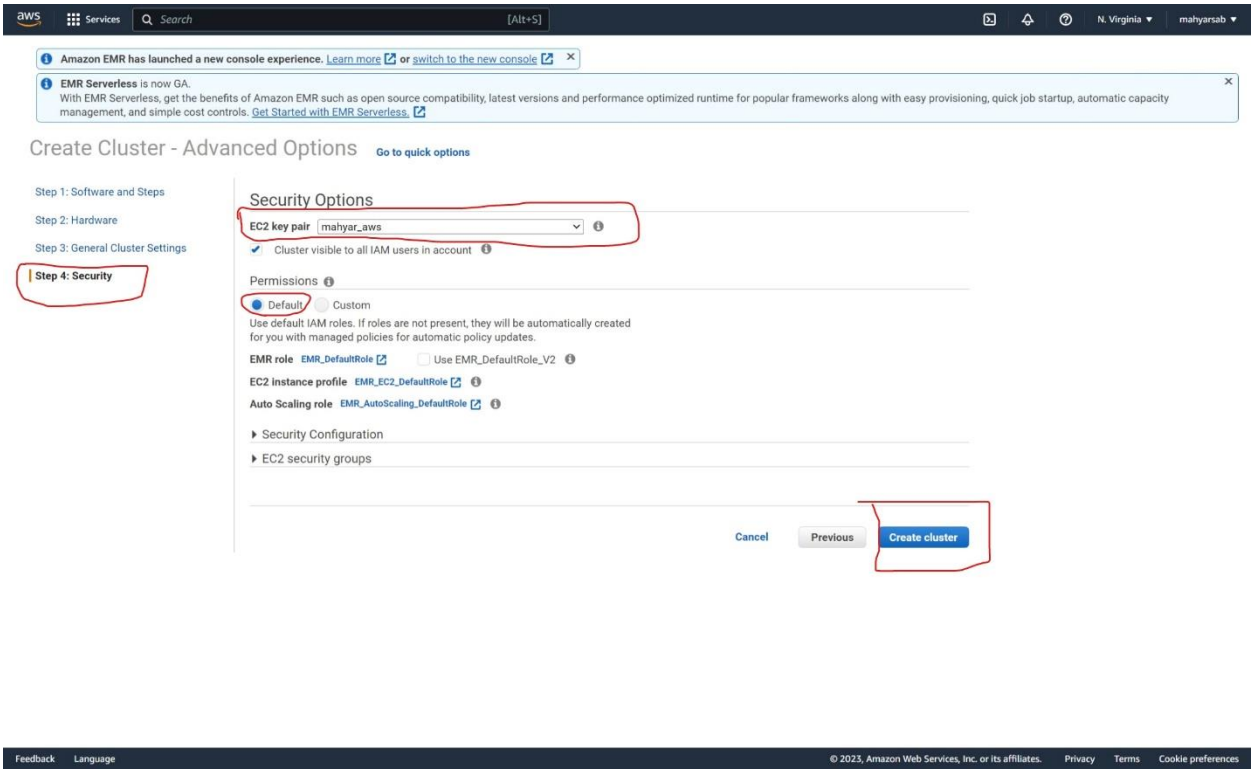
- Choose the abovementioned items in first step:



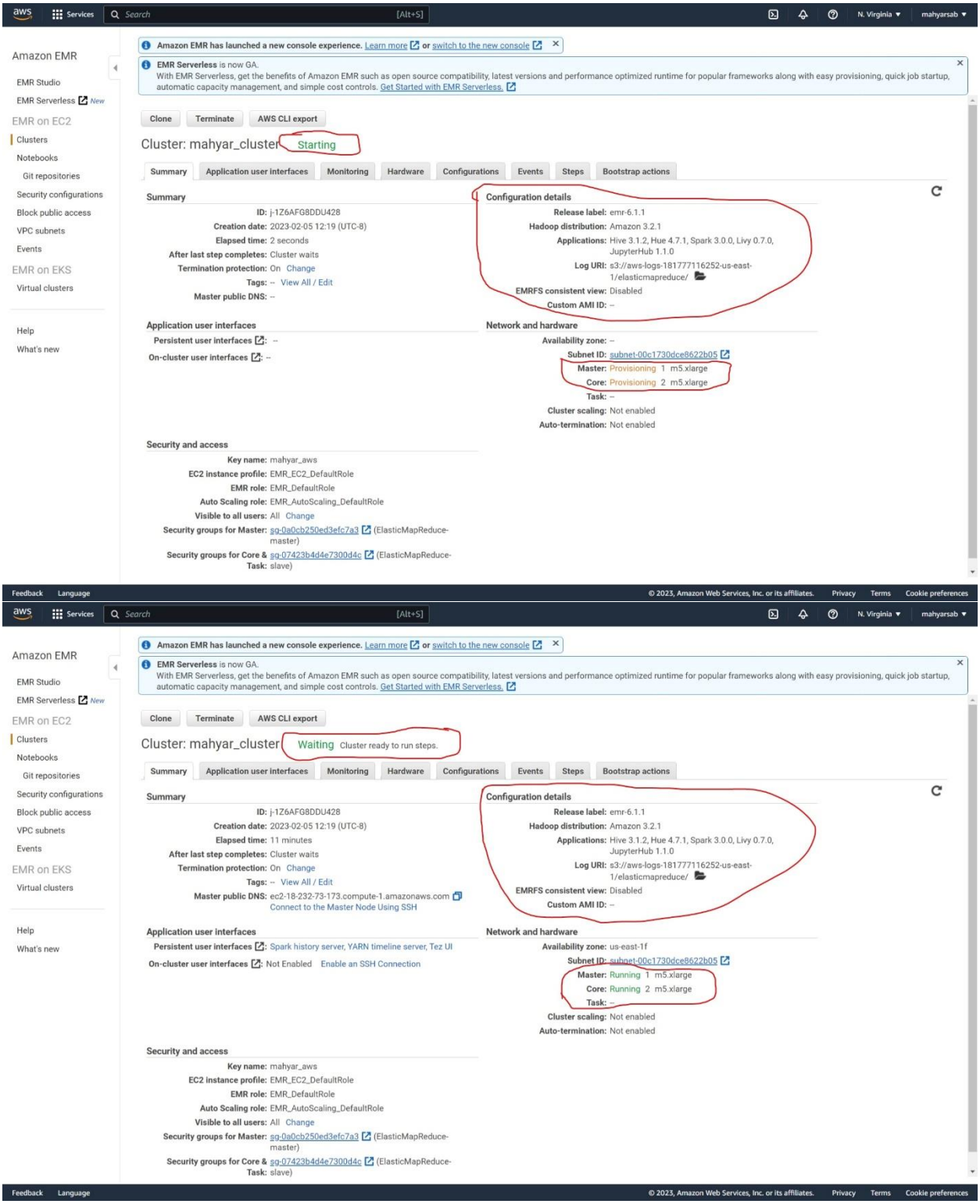
- Choose a name for your cluster in step 3:



- Choose your AWS EC2 key in step4: Security and then click on Create cluster:

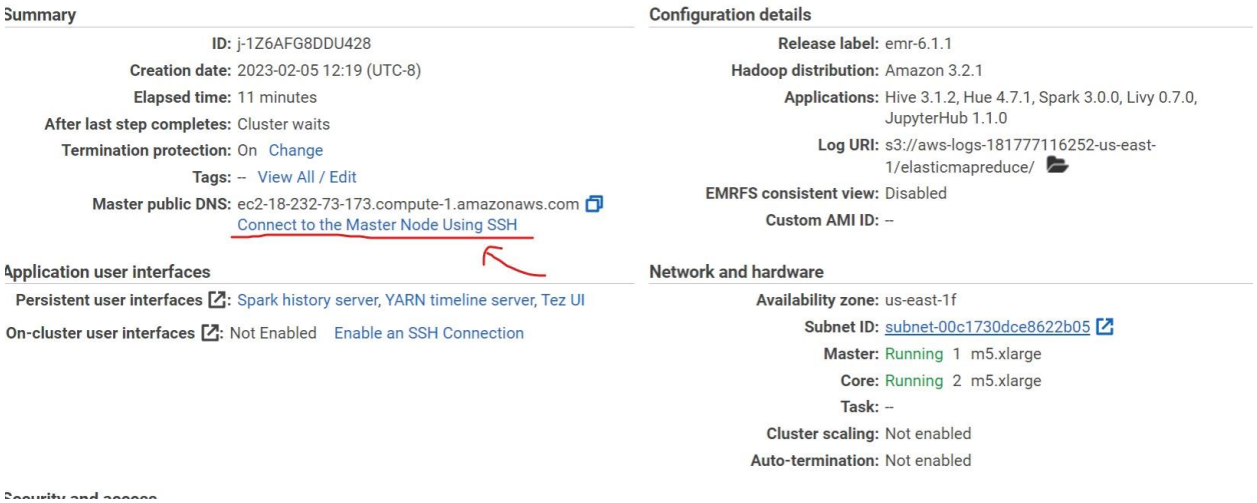


- Then the cluster shows starting status and after a while it will change to Waiting:

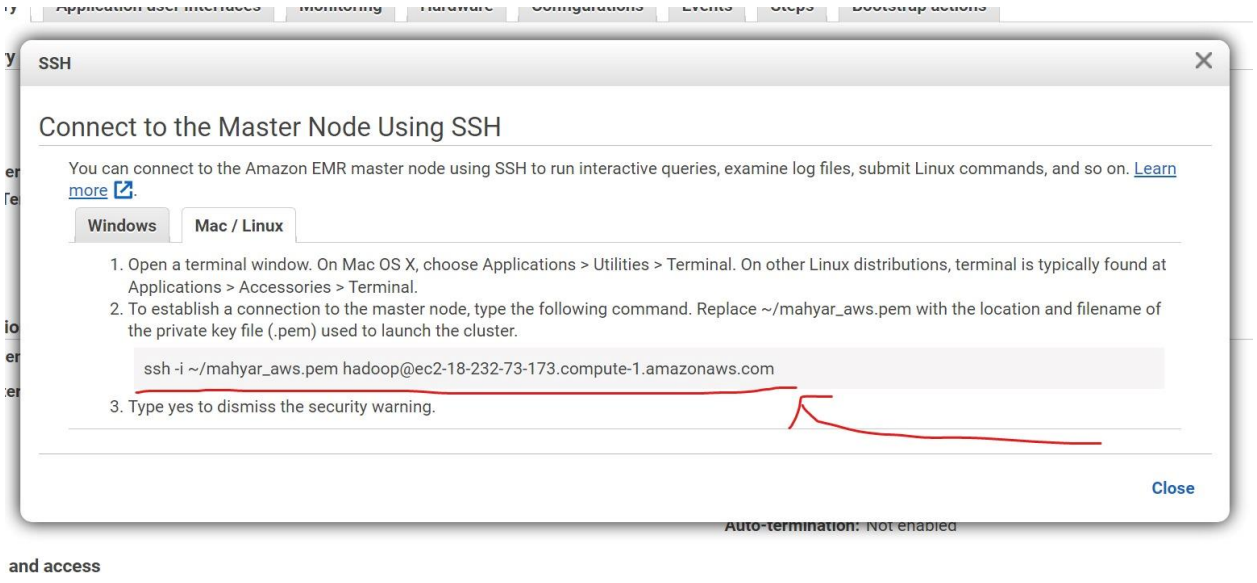


- Now the cluster is ready to use.

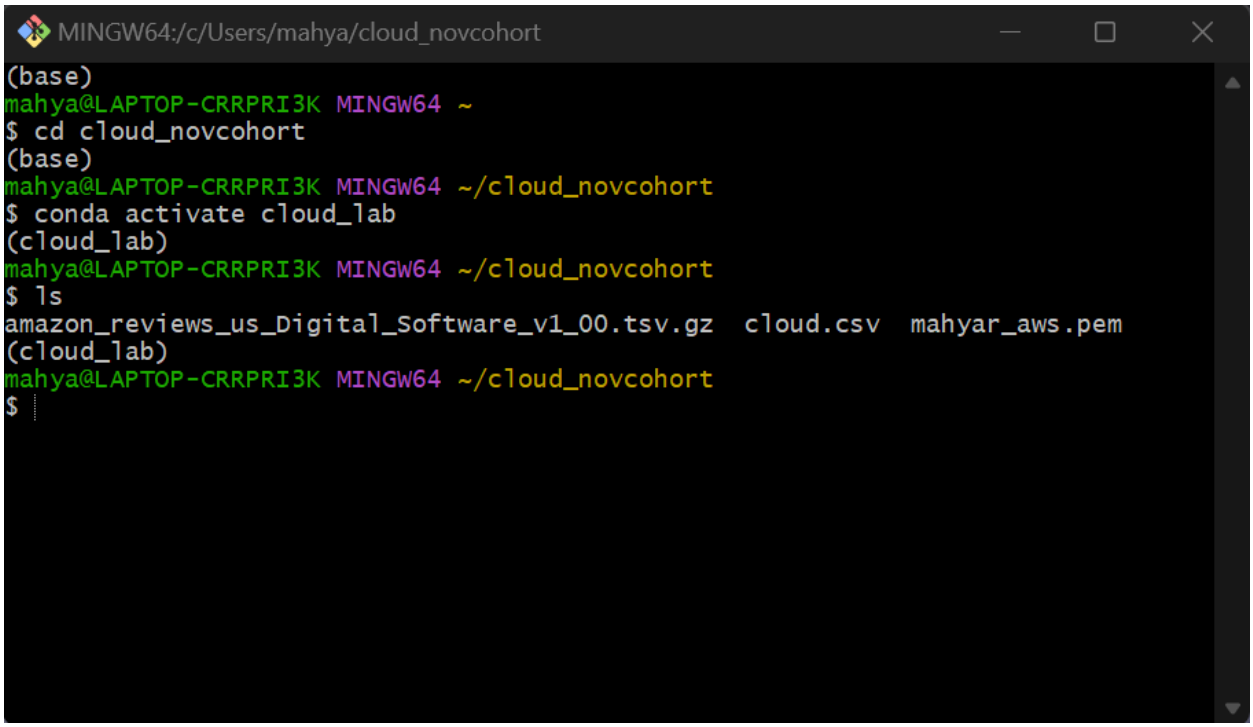
Question 2: In this question we should connect to the head node of the cluster using SSH:



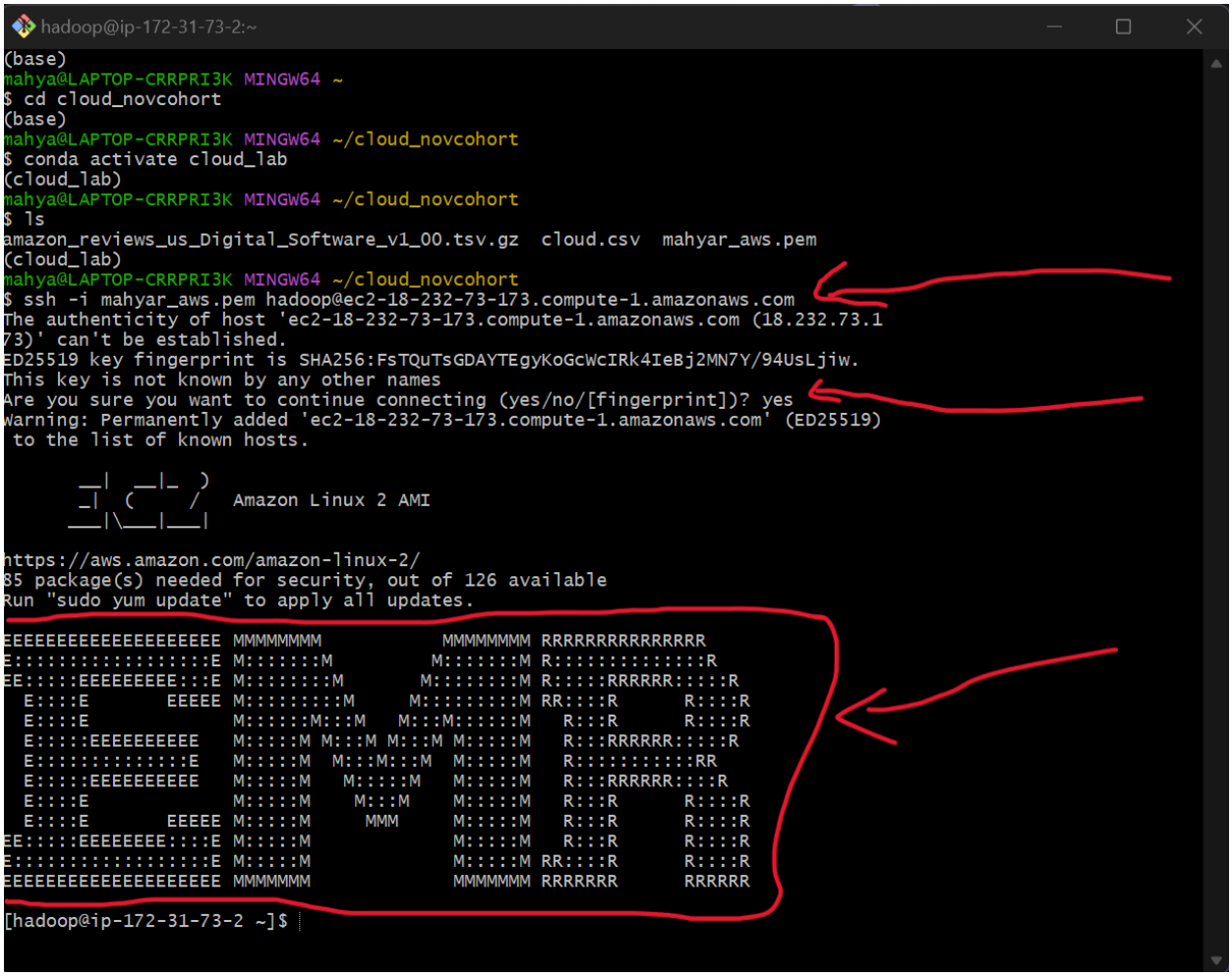
- Click on the mentioned button and copy the underlined part starting with SSH:



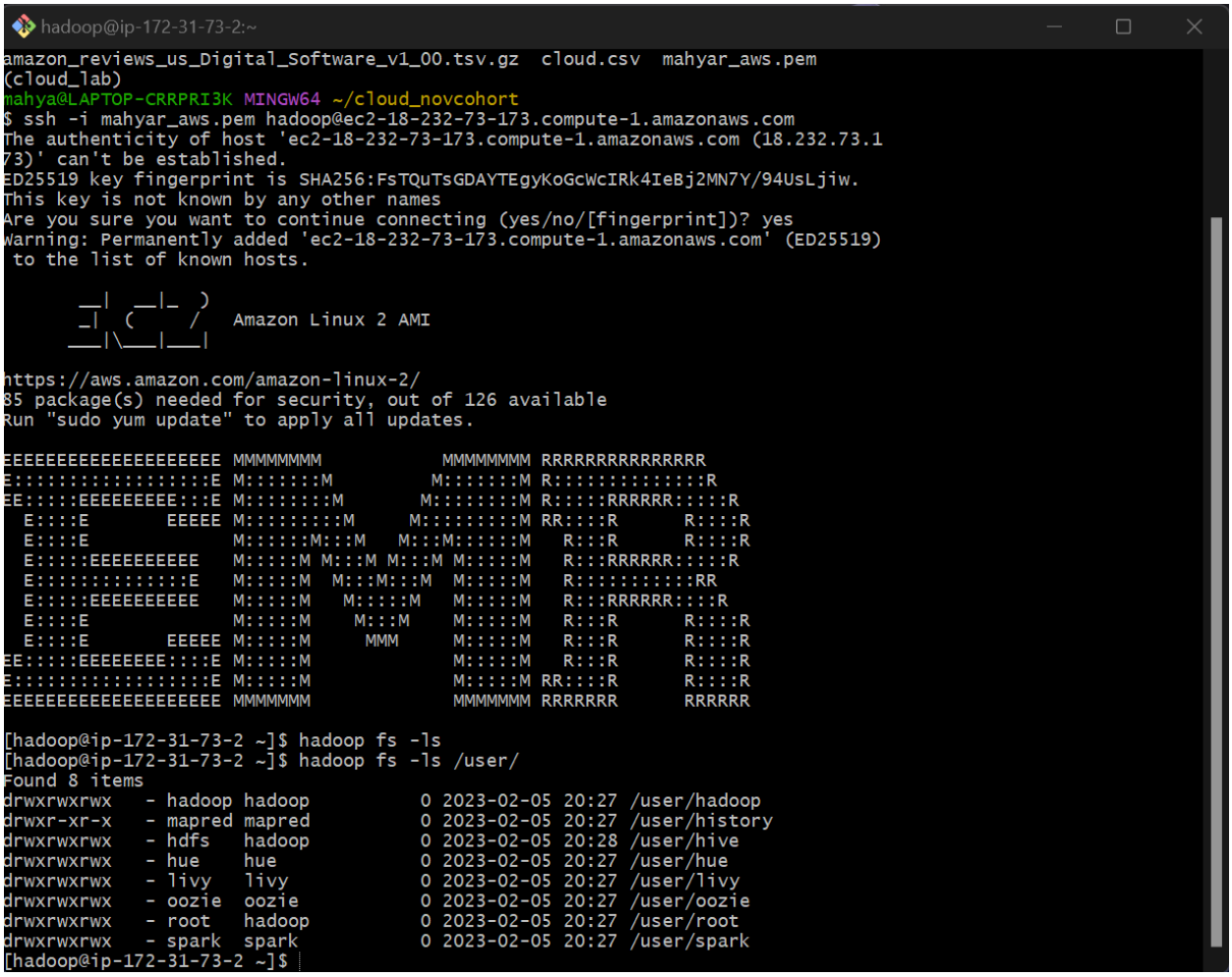
- Now open your terminal and change directory to your cloud folder which contains your pem key which mine here is mahyar_aws.pem :



- Now paste the copied ssh code from amazon website here in your terminal and delete ~/ from the beginning of the name of your pem key and execute it. Then type yes and you will see a big EMR on your terminal and now you are connected:



- Now let’s execute some Hadoop commands here:

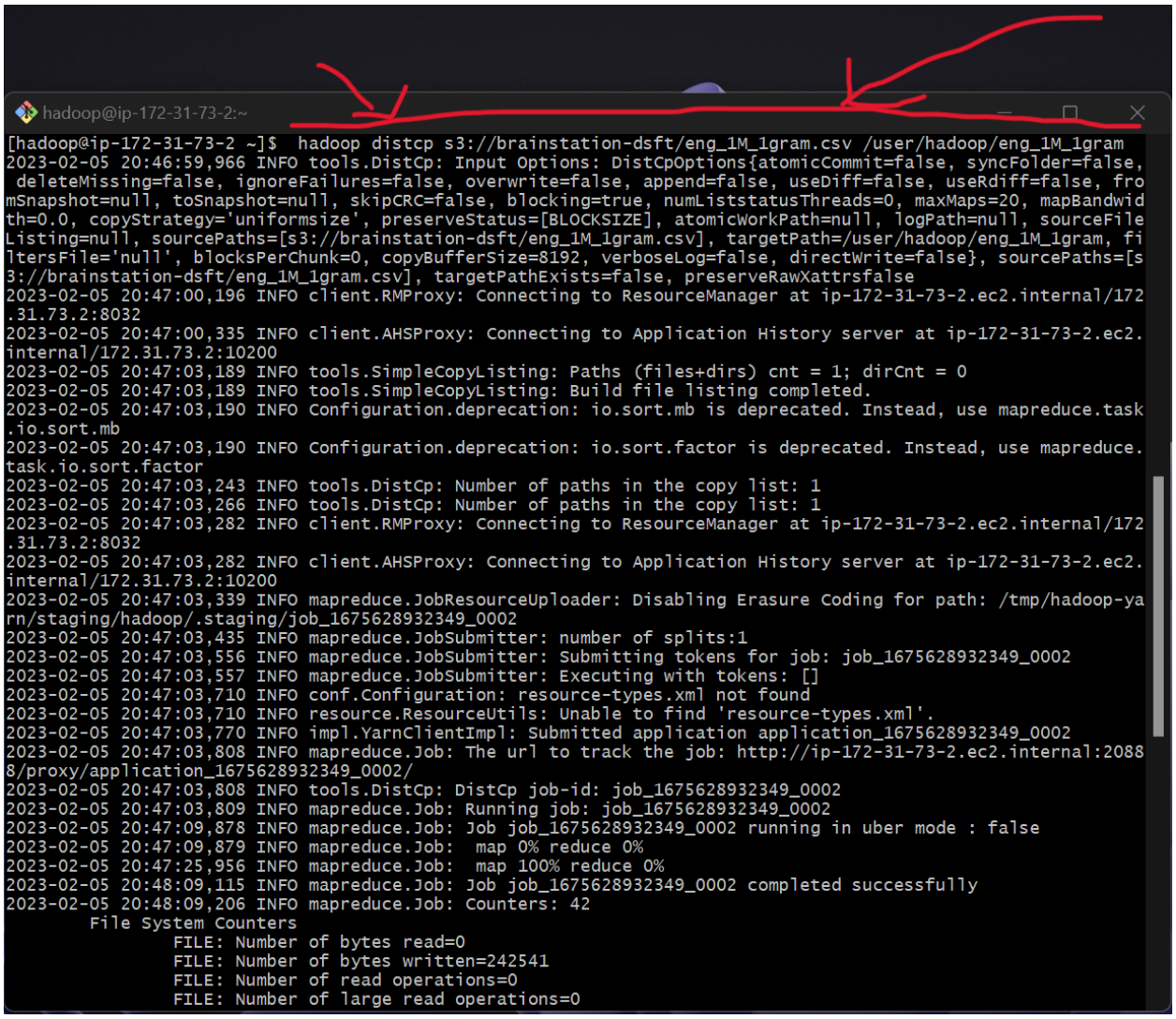


QUESTION 3 : We will copy the data folder from the S3 bucket *directly* into a directory on the Hadoop File System (HDFS) named /user/hadoop/eng_1M_1gram using the underlined code below:

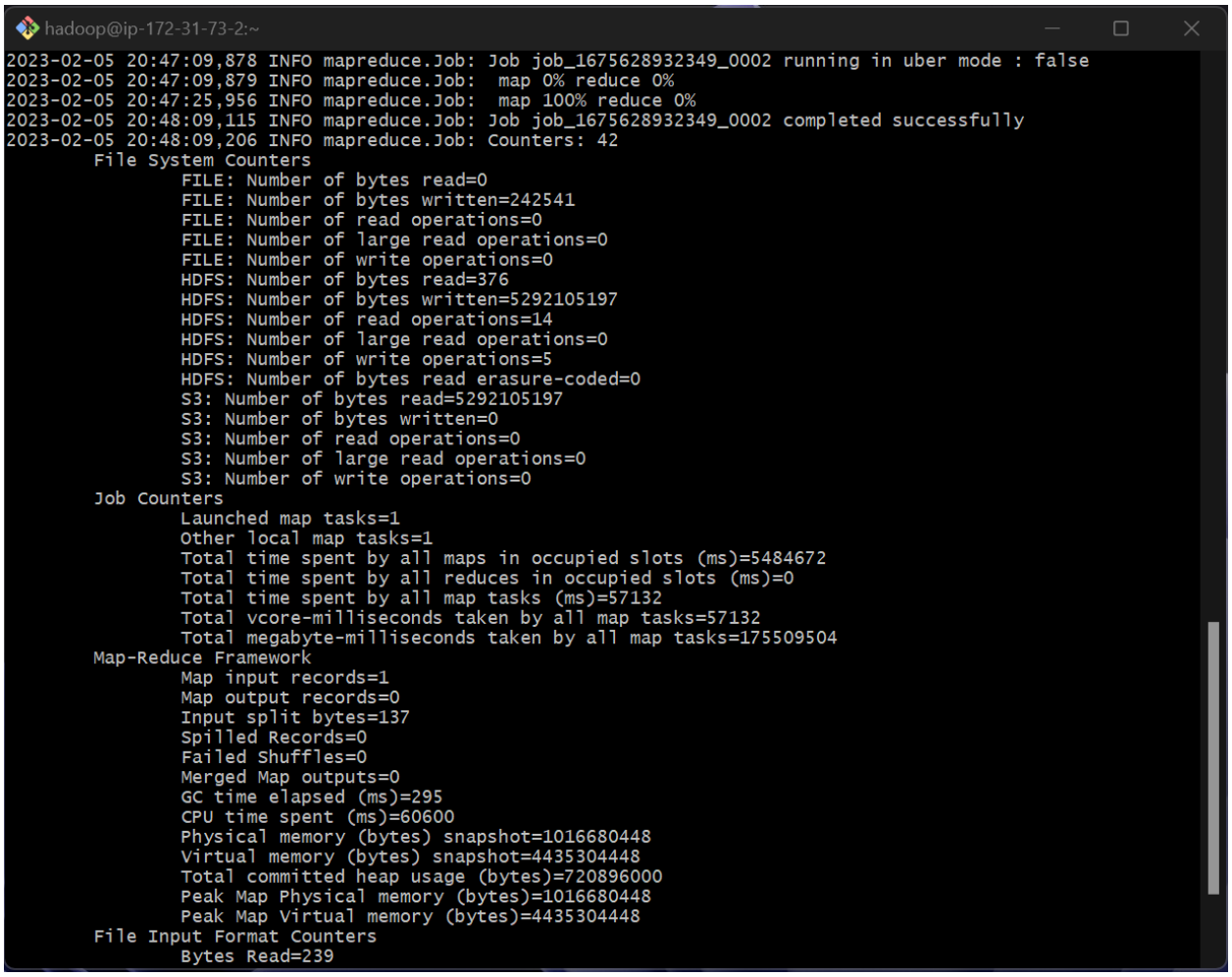
- In this question I used 'hadoop distcp' instead of cp for the reason below:

  hadoop distcp is a Hadoop command-line tool that can be used to copy large amounts of data between Hadoop clusters, or between a Hadoop cluster and an external storage system such as a local file system or an Amazon S3 bucket.
  hadoop distcp uses MapReduce to perform the data transfer, which allows it to scale to handle very large data sets and to recover from failures. The tool also supports various advanced features, such as preserving file attributes, copying only new or updated files, and throttling the data transfer rate.

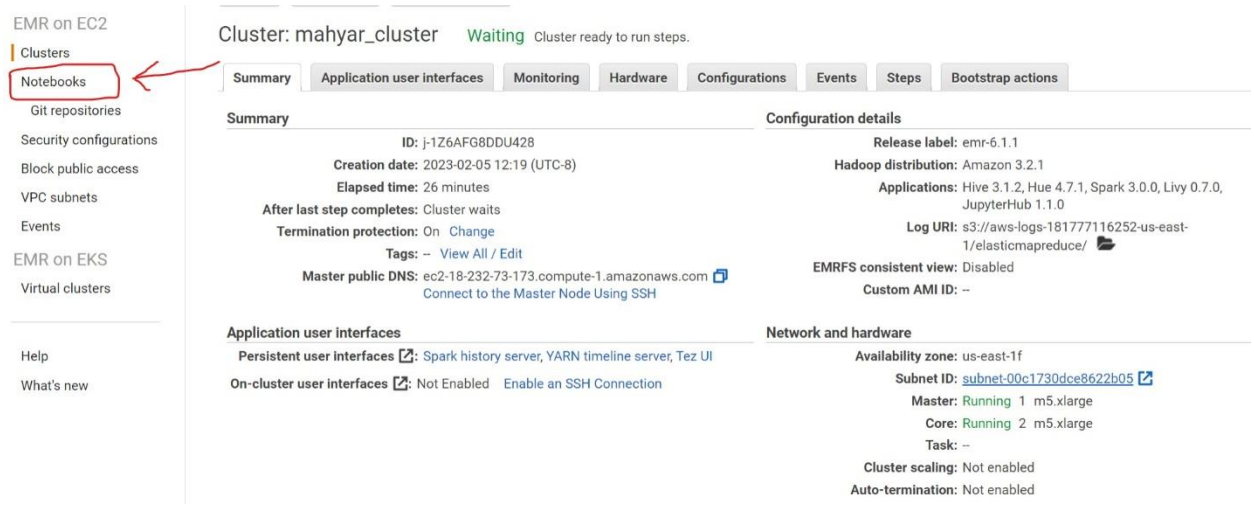
hadoop distcp is often used for large-scale data migration, data backup and recovery, and data sharing between Hadoop clusters. It is a powerful tool for managing big data, but it can also be complex to use, so it is important to understand the various options and configuration settings available when using **hadoop distcp**.

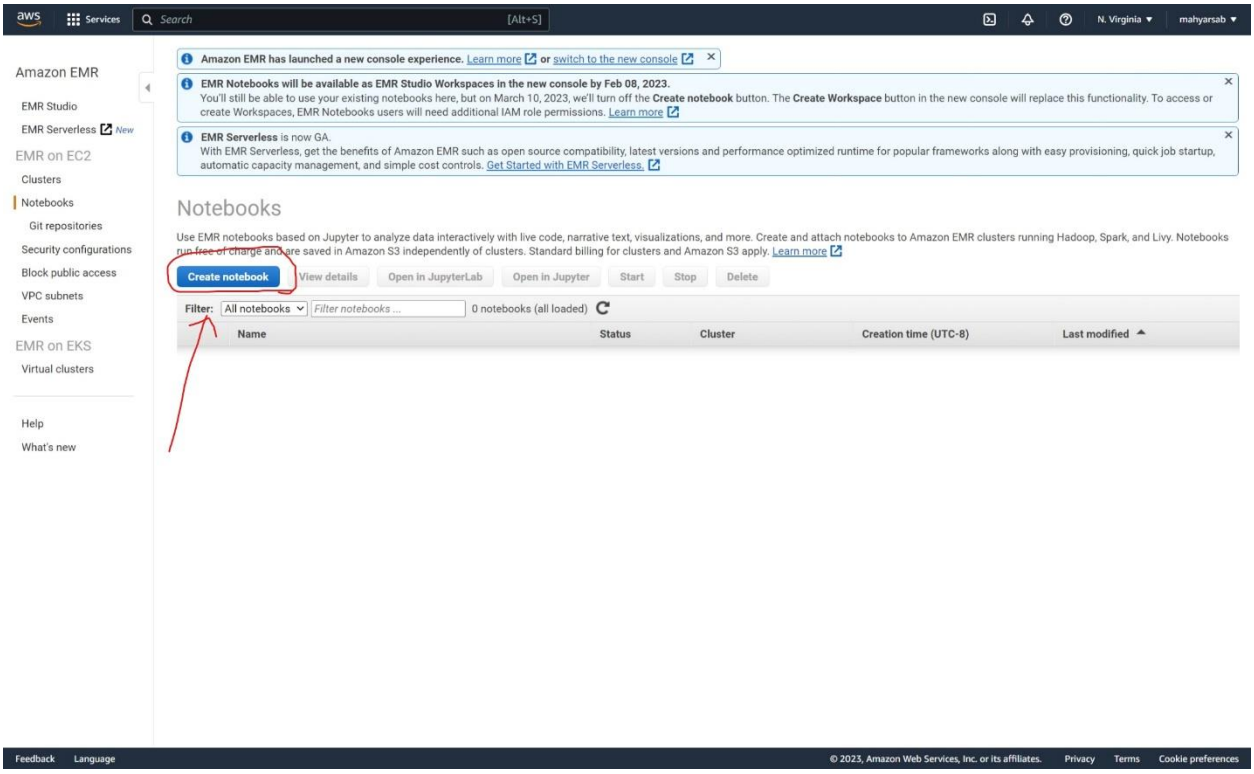


- After that we will see that it will copy and we see the executions below:

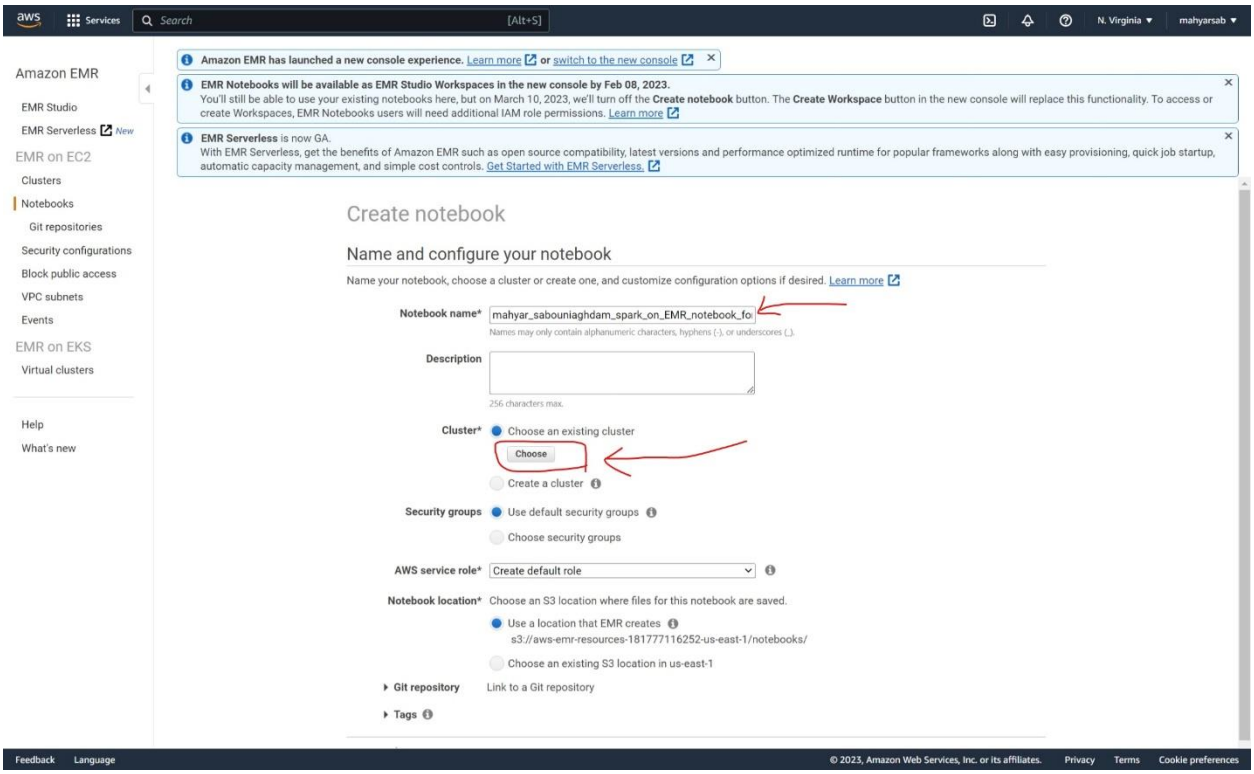


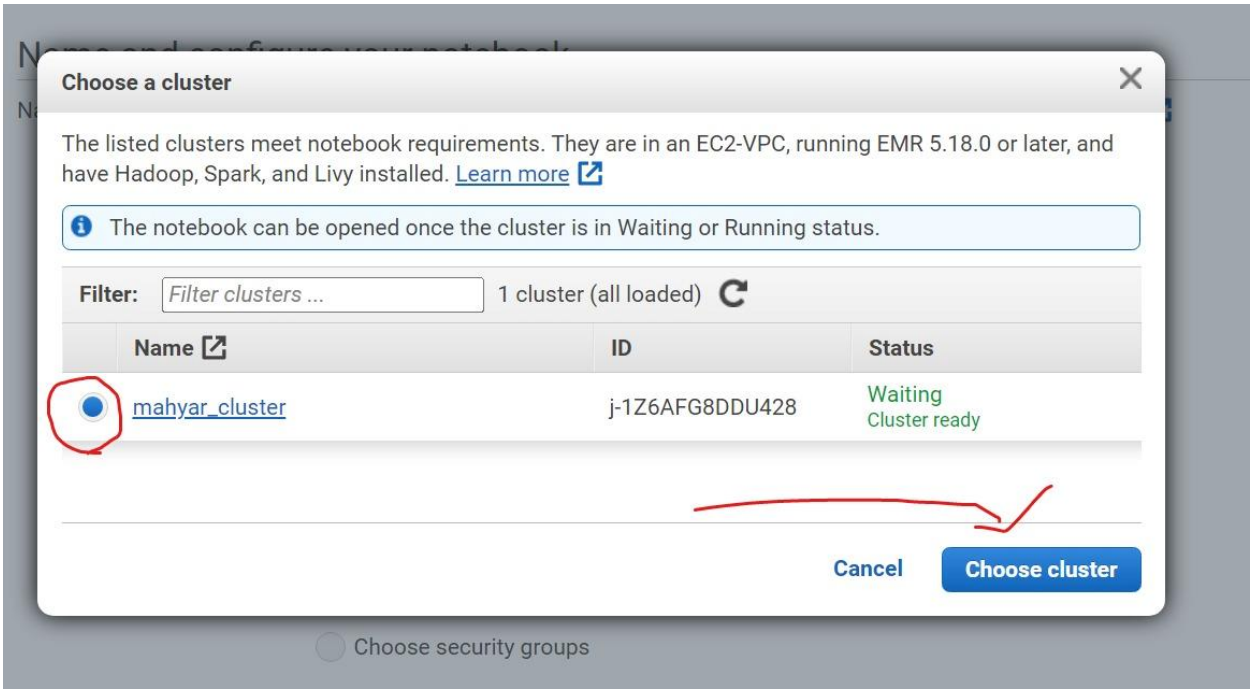
Question 4 : Now by using pyspark, we have to read the data that we copied into HDFS in Step 3. For this I will Launch a jupyter notebook from amazon aws like below:

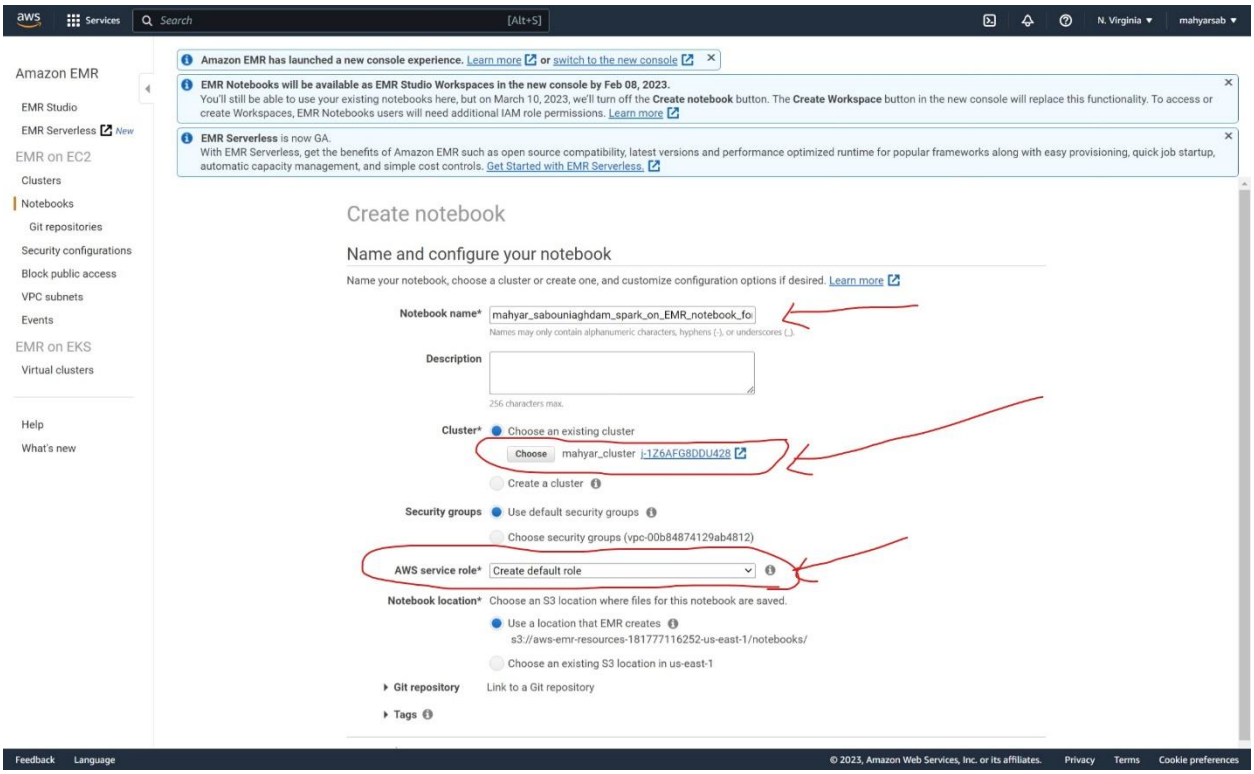




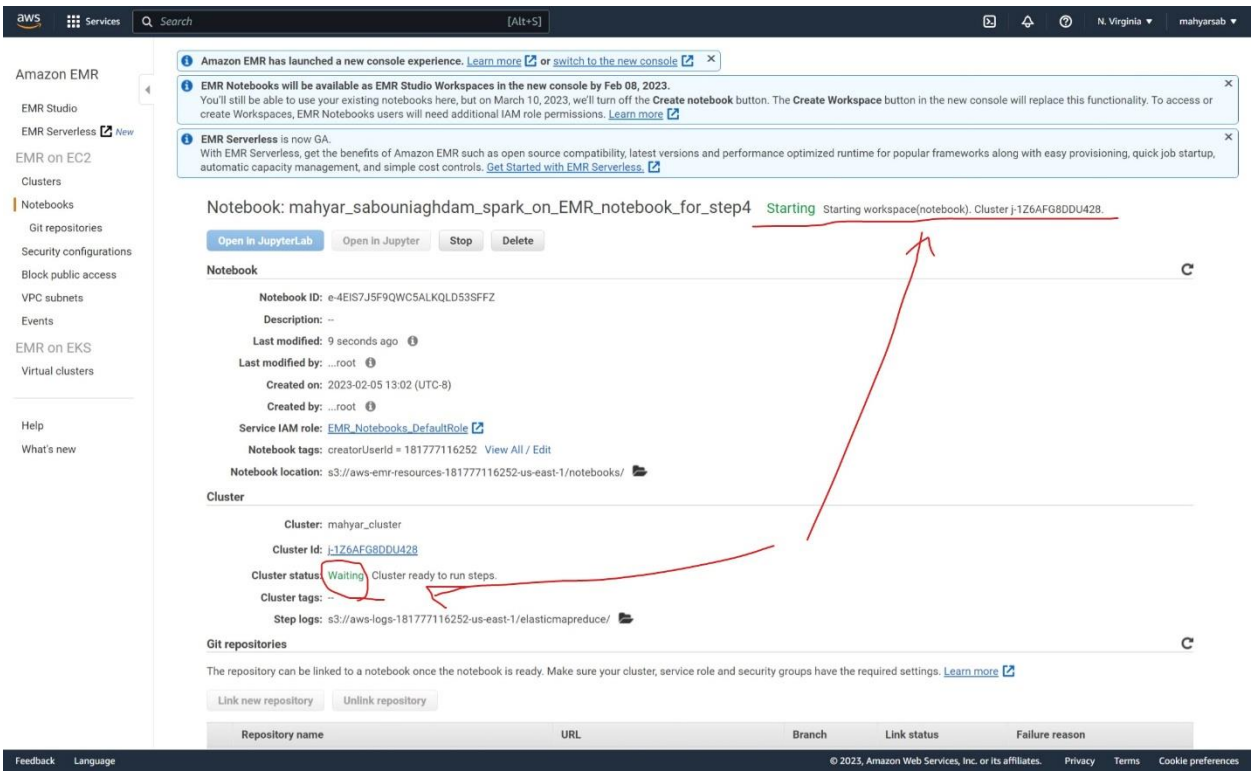
- Here we should give the notebook a name then choose our cluster and then put AWS service role on create default role and the click on create notebook:


aws
Services
Search
[Alt+S]
N. Virginia
mahyarsab
Amazon EMR
EMR Studio
EMR Serverless New
EMR on EC2
Clusters
Notebooks
Git repositories
Security configurations
Block public access
VPC subnets
Events
EMR on EKS
Virtual clusters
Help
What's new
Amazon EMR has launched a new console experience. Learn more or switch to the new console
EMR Notebooks will be available as EMR Studio Workspaces in the new console by Feb 08, 2023.
You'll still be able to use your existing notebooks here, but on March 10, 2023, we'll turn off the Create notebook button. The Create Workspace button in the new console will replace this functionality. To access or create Workspaces, EMR Notebooks users will need additional IAM role permissions. Learn more
EMR Serverless is now GA.
With EMR Serverless, get the benefits of Amazon EMR such as open source compatibility, latest versions and performance optimized runtime for popular frameworks along with easy provisioning, quick job startup, automatic capacity management, and simple cost controls. Get Started with EMR Serverless.
Create notebook
Name and configure your notebook
Name your notebook, choose a cluster or create one, and customize configuration options if desired. Learn more
Notebook name*
mahyar_sabouniaghdam_spark_on_EMR_notebook_fo
Names may only contain alphanumeric characters, hyphens (-), or underscores (_).
Description
256 characters max.
Cluster*
Choose an existing cluster
Choose
Create a cluster
Security groups
Use default security groups
Choose security groups
AWS service role*
Create default role
Notebook location*
Choose an S3 location where files for this notebook are saved.
Use a location that EMR creates
s3://aws-emr-resources-181777116252-us-east-1/notebooks/
Choose an existing S3 location in us-east-1
Git repository
Link to a Git repository
Tags
Feedback
Language
© 2023, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

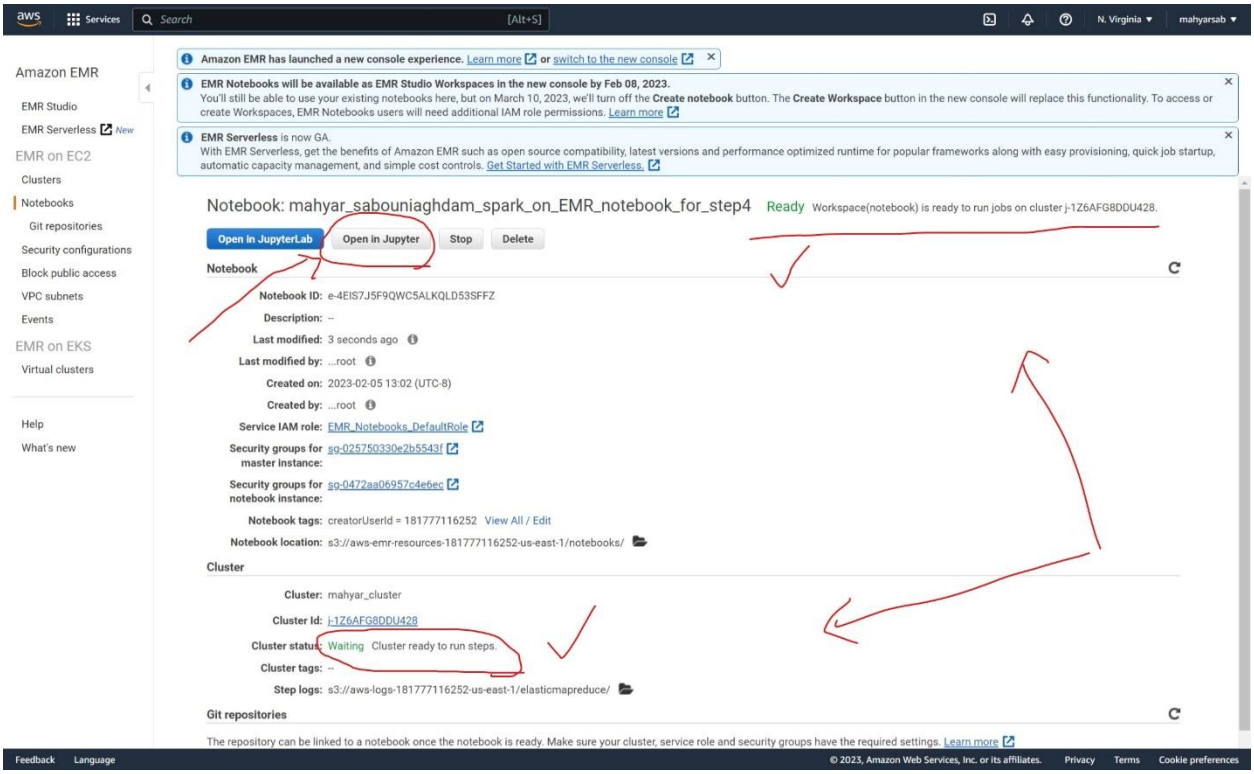

Choose a cluster
The listed clusters meet notebook requirements. They are in an EC2-VPC, running EMR 5.18.0 or later, and have Hadoop, Spark, and Livy installed. Learn more
The notebook can be opened once the cluster is in Waiting or Running status.
Filter:
Filter clusters ...
1 cluster (all loaded)
Name
ID
Status
mahyar_cluster
j-1Z6AFG8DDU428
Waiting
Cluster ready
Cancel
Choose cluster
Choose security groups



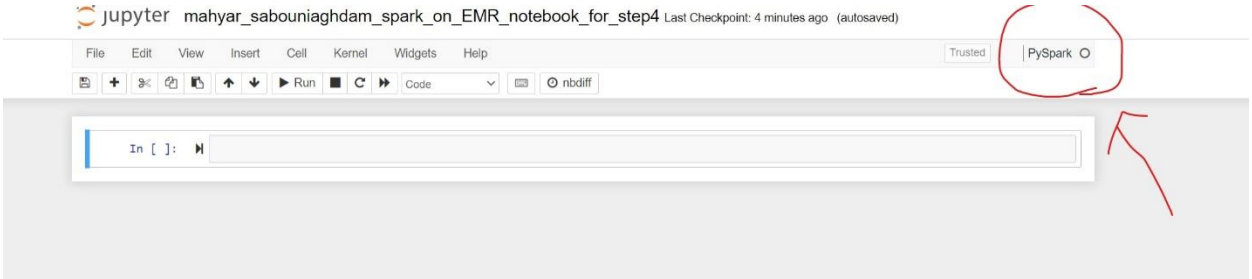
- First our notebook status is starting and after a while it will change to Ready.



- Click on open in jupyter and we will continue our work in jupyter notebook:



- Our kernel should be PySpark:



Question 4: the whole complete answer for step 4 is on the notebook attached. I will share the pictures here too:

jupyter mahyar_sabouniaghdam_spark_on_EMR_notebook_for_step4 Last Checkpoint: a few seconds ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted PySpark

Run Markdown nbdiff

Mahyar Sabouniaghdam

BrainStation®

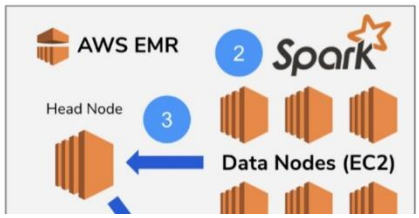
Big Data Wrangling With Google Books Ngrams

In this assignment, we will apply the skills we've learned in the Big Data Fundamentals unit to load, filter, and visualize a large real-world dataset in a cloud-based distributed computing environment using Hadoop, Spark, Hive, and the S3 filesystem. We will prepare a professional report to summarize the findings and be sure to include an appendix with screenshots of the steps completed for Questions 1 and 2.

The Google Ngrams dataset was created by Google's research team by analyzing all of the content in Google Books - these digitized texts represent approximately 4% of all books ever printed, and span a time period from the 1800s into the 2000s.

The dataset is hosted in a public S3 bucket as part of the Amazon S3 Open Data Registry. For this assignment, we have converted the data to CSV and hosted it on a public S3 bucket which may be accessed here: s3://brainstation-dsft/eng_1M_1gram.csv

For this deliverable, we will produce a report, as well as a jupyter notebook, which will follow a Big Data analysis workflow. As part of this workflow we will filter and reduce data down to a manageable size, and then do some analysis locally on our machine after extracting data from the Cloud and processing it using Big Data tools. The workflow and steps in the process are illustrated below:



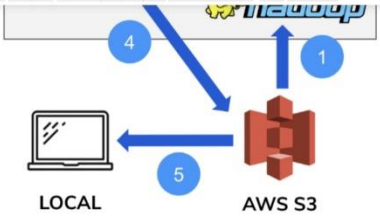
Big Data Analysis Workflow

1. Pull data from (public) S3 bucket into HDFS
2. Analyze and filter data using Spark

jupyter mahyar_sabouniaghdam_spark_on_EMR_notebook_for_step4 Last Checkpoint: a few seconds ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted PySpark

Run Markdown nbdiff



3. node using *getmerge*
4. Write data to (personal) S3 bucket
5. Read data from S3 bucket for analysis on local machine

1. Up untill now we spined up a new EMR cluster using the AWS Console and we included Hadoop, Spark, Hive, Jupyterhub, and Livy for our cluster with the release version of EMR 6.1.1.
2. Then we connected to the head node of the cluster using SSH.
3. Then we copied the data folder from s3://brainstation-dsft/eng_1M_1gram.csv into a directory on the Hadoop File System (HDFS) named /user/hadoop/eng_1M_1gram.
4. Now we are in step 4:

Step 4

In this step by using pyspark, we will read the data we copied into HDFS in Step 3 with a PySpark notebook that we created. Since we have created a pyspark DataFrame and now we are in it, we will complete the following steps below:

a. Describe the dataset (examples include size, shape, schema) in pyspark.

b. Create a new DataFrame from a query using Spark SQL, filtering to include only the rows where the token is "data" and describe the new dataset.

c. Write the filtered data back to a directory in the HDFS from Spark using df.write.csv(). Be sure to pass the header=True parameter and examine the contents of what you've written.

- First we have to initialize the SparkContext and SparkSession:

In [1]:
```
spark
```

```
Starting Spark application
```

Step 4

In this step by using pyspark, we will read the data we copied into HDFS in Step 3 with a PySpark notebook that we created. Since we have created a pyspark DataFrame and now we are in it, we will complete the following steps below:

a. Describe the dataset (examples include size, shape, schema) in pyspark.

b. Create a new DataFrame from a query using Spark SQL, filtering to include only the rows where the token is "data" and describe the new dataset.

c. Write the filtered data back to a directory in the HDFS from Spark using df.write.csv(). Be sure to pass the header=True parameter and examine the contents of what you've written.

- First we have to initialize the SparkContext and SparkSession:

In [1]:
```
spark
```

```
Starting Spark application
```

ID	YARN Application ID	Kind	State	Spark UI	Driver log	Current session?
0	application_1675628932349_0003	pyspark	idle	Link	Link	✓

```
SparkSession available as 'spark'.

<pyspark.sql.session.SparkSession object at 0x7f09c79955d0>
```

- Now we will read the data:

In [3]:
```
# read the data

data = spark.read.csv("/user/hadoop/eng_1M_1gram", header=True)
```

▾ Spark Job Progress

▾ Job [1]: csv at NativeMethodAccessorImpl.java:0

Progress for csv at NativeMethodAccessorImpl.java:0 Job Progress: 1/1 Tasks Complete

Stage [ID]: name at [source]:[line]	Status	Task Progress	Elapsed Time (seconds)	Failed Task Logs
Stage [1]: csv at NativeMet...java:0	COMPLETE	1/1	7.532	

jupyter mahyar_sabouniaghdam_spark_on_EMR_notebook_for_step4 Last Checkpoint: a minute ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted PySpark

Run Markdown nbdiff

4.a

- Now we will describe the dataset (examples include size, shape, schema) in pyspark:

In [4]:
```
# Get the shape of the dataset

print("Number of rows:", data.count())
print("Number of columns:", len(data.columns))
```

▸ Spark Job Progress

```
Number of rows: 261823225
Number of columns: 5
```

- Our dataset has 261823225 rows and 5 columns. Let's see the columns our dataset:

In [19]:
```
# columns

data.columns
```

```
['token', 'year', 'frequency', 'pages', 'books']
```

- Now let's see the schema of our dataset:

In [5]:
```
# Get the schema of the dataset

data.printSchema()
```

```
root
 |-- token: string (nullable = true)
 |-- year: string (nullable = true)
 |-- frequency: string (nullable = true)
 |-- pages: string (nullable = true)
 |-- books: string (nullable = true)
```

- Let's see the first 20 rows of the dataset:

```
data.printSchema()
```

```
root
 |-- token: string (nullable = true)
 |-- year: string (nullable = true)
 |-- frequency: string (nullable = true)
 |-- pages: string (nullable = true)
 |-- books: string (nullable = true)
```

- Let's see the first 20 rows of the dataset:

In [20]:

```
# First 20 rows

data.show(20)
```

```
+---------+----+---------+-----+-----+
|    token|year|frequency|pages|books|
+---------+----+---------+-----+-----+
|inGermany|1927|        2|    2|    2|
|inGermany|1929|        1|    1|    1|
|inGermany|1930|        1|    1|    1|
|inGermany|1933|        1|    1|    1|
|inGermany|1934|        1|    1|    1|
|inGermany|1935|        1|    1|    1|
|inGermany|1938|        5|    5|    5|
|inGermany|1939|        1|    1|    1|
|inGermany|1940|        1|    1|    1|
|inGermany|1942|        2|    2|    2|
|inGermany|1944|        2|    2|    2|
|inGermany|1946|        2|    2|    2|
|inGermany|1947|        3|    3|    2|
|inGermany|1948|        1|    1|    1|
|inGermany|1949|        1|    1|    1|
|inGermany|1952|        1|    1|    1|
|inGermany|1956|        1|    1|    1|
|inGermany|1957|        2|    2|    2|
|inGermany|1959|        1|    1|    1|
|inGermany|1960|        3|    3|    3|
+---------+----+---------+-----+-----+
only showing top 20 rows
```

4.b

jupyter mahyar_sabouniaghdam_spark_on_EMR_notebook_for_step4 Last Checkpoint: a minute ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted PySpark

Run Markdown nbdiff

4.b

- Now we will create a new DataFrame from a query using Spark SQL, filtering to include only the rows where the token is "data" and then we will describe the new dataset:

- Using SQL requires the CreateorReplaceTempView function, which registers the data as a view in the Spark session. We can then query against that view with SQL, with the view name being the table name:

In [21]:

```
# Create a view

data.createOrReplaceTempView("ngrams")
```

In [22]:

```
# Create a new DataFrame from a query using Spark SQL

filtered_data = spark.sql("SELECT * FROM ngrams WHERE token = 'data'")
```

- Now we created a new DataFrame (filtered_data) using Spark SQL and filtered it to include only the rows where the token is "data". Now let's describe the new dataset:

In [23]:

```
# Get the shape of the filtered dataset

print("Number of rows:", filtered_data.count())
print("Number of columns:", len(filtered_data.columns))
```

```
Number of rows: 316
Number of columns: 5
```

- Our dataset has 316 rows and 5 columns. Let's see the columns our dataset:

In [27]:

```
# columns

filtered_data.columns
```

```
['token', 'year', 'frequency', 'pages', 'books']
```

In [28]:

```
# Get the schema of the dataset
```

- Our dataset has 316 rows and 5 columns. Let's see the columns our dataset:

In [27]:
```
# columns

filtered_data.columns
```

```
['token', 'year', 'frequency', 'pages', 'books']
```

In [28]:
```
# Get the schema of the dataset

filtered_data.printSchema()
```

```
root
 |-- token: string (nullable = true)
 |-- year: string (nullable = true)
 |-- frequency: string (nullable = true)
 |-- pages: string (nullable = true)
 |-- books: string (nullable = true)
```

- We saw the columns and schema of the filtered dataset again. Let's see the first 20 rows of the filtered dataset:

In [29]:
```
# First 20 rows of the filtered dataset

filtered_data.show(20)
```

```
+-----+----+---------+-----+-----+
|token|year|frequency|pages|books|
+-----+----+---------+-----+-----+
| data|1584|       16|   14|    1|
| data|1614|        3|    2|    1|
| data|1627|        1|    1|    1|
| data|1631|       22|   18|    1|
| data|1637|        1|    1|    1|
| data|1638|        2|    2|    1|
| data|1640|        1|    1|    1|
| data|1642|        1|    1|    1|
| data|1644|        4|    4|    1|
| data|1647|        1|    1|    1|
| data|1651|        1|    1|    1|
| data|1674|        1|    1|    1|
| data|1690|        1|    1|    1|
| data|1693|        1|    1|    1|
| data|1697|        1|    1|    1|
| data|1699|        1|    1|    1|
| data|1700|        1|    1|    1|
```

jupyter mahyar_sabouniaghdam_spark_on_EMR_notebook_for_step4 Last Checkpoint: 2 minutes ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help Trusted PySpark

Run Markdown nbdiff

```
| data|1637|        1|    1|    1|
| data|1638|        2|    2|    1|
| data|1640|        1|    1|    1|
| data|1642|        1|    1|    1|
| data|1644|        4|    4|    1|
| data|1647|        1|    1|    1|
| data|1651|        1|    1|    1|
| data|1674|        1|    1|    1|
| data|1690|        1|    1|    1|
| data|1693|        1|    1|    1|
| data|1697|        1|    1|    1|
| data|1699|        1|    1|    1|
| data|1700|        1|    1|    1|
| data|1701|       11|   11|    2|
| data|1702|        1|    1|    1|
| data|1703|        1|    1|    1|
+-----+----+---------+-----+-----+
only showing top 20 rows
```

- Number of rows of our filtered dataset:

In [30]:
```
# Number of rows of filtered data

filtered_data.count()
```

```
316
```

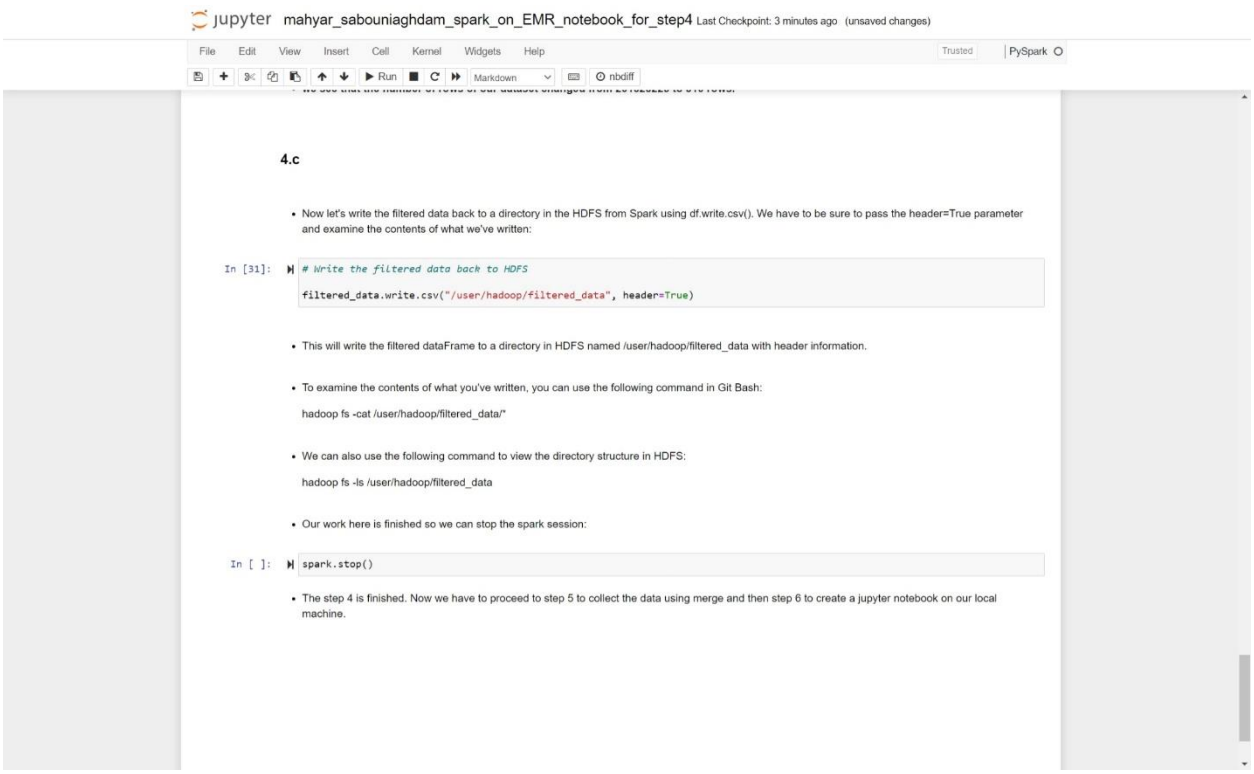
- **we see that the number of rows of our dataset changed from 261823225 to 316 rows.**

4.c

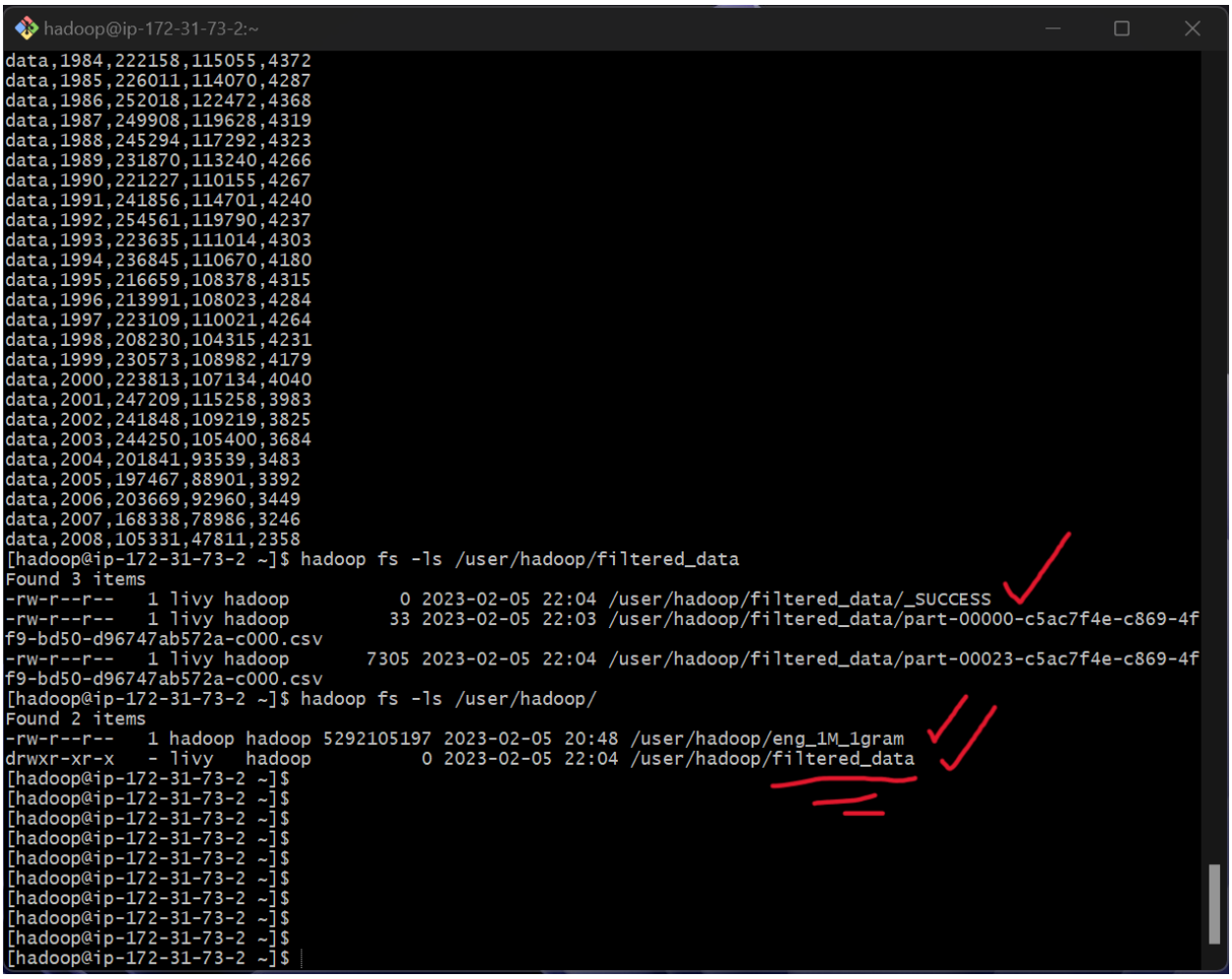
- Now let's write the filtered data back to a directory in the HDFS from Spark using df.write.csv(). We have to be sure to pass the header=True parameter and examine the contents of what we've written:

In [31]:
```
# Write the filtered data back to HDFS

filtered_data.write.csv("/user/hadoop/filtered_data", header=True)
```

- This will write the filtered dataFrame to a directory in HDFS named /user/hadoop/filtered_data with header information.



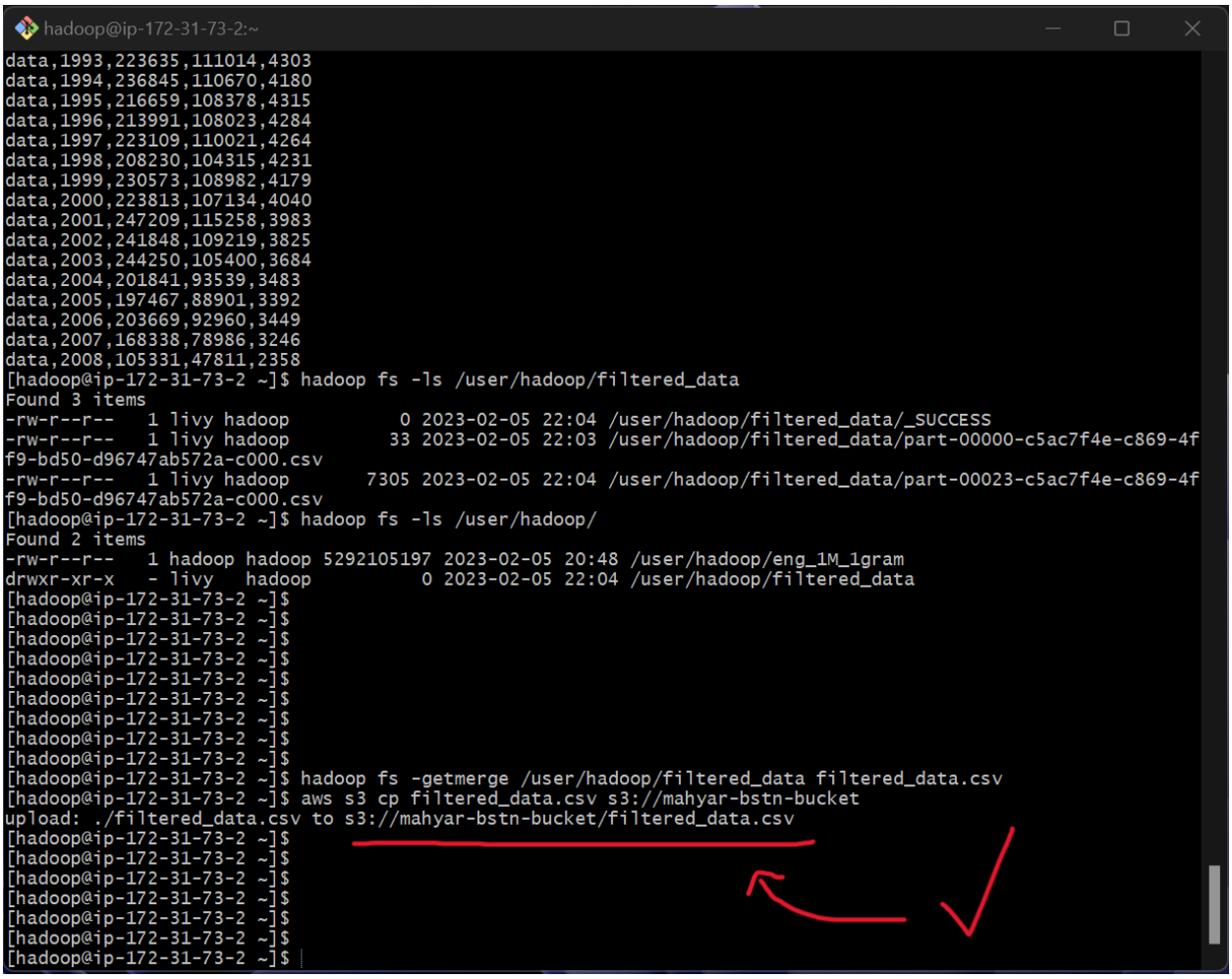
- Now that we wrote the filtered_data back to HDFS, let's run some codes on command line to check whether they exist or not:



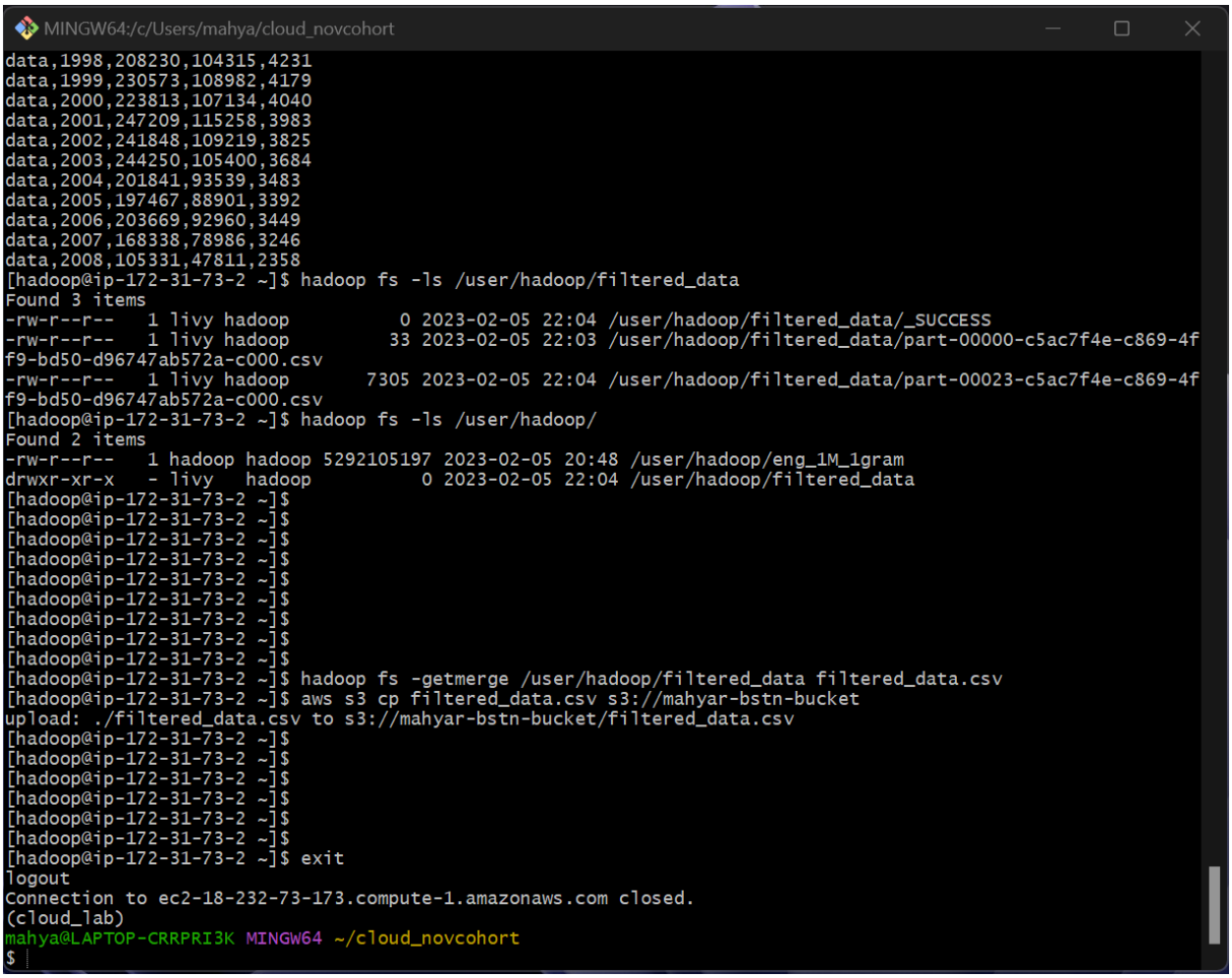
- We can see that our filtered_data.csv file is on Hadoop.

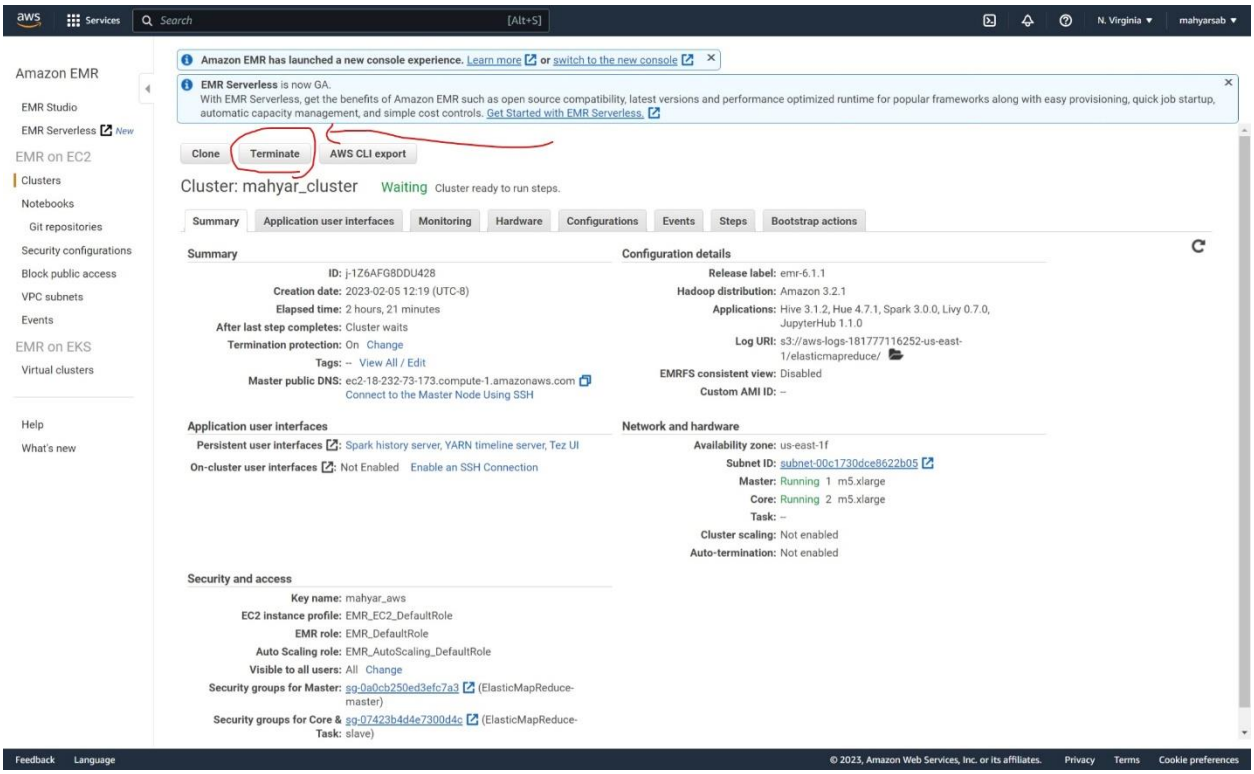
Question 5 : In this step we collect the contents of the directory into a single file on the local drive of the head node using getmerge and then we moved filtered file into S3 bucket of my account which is mahyar-bstn-bucket:

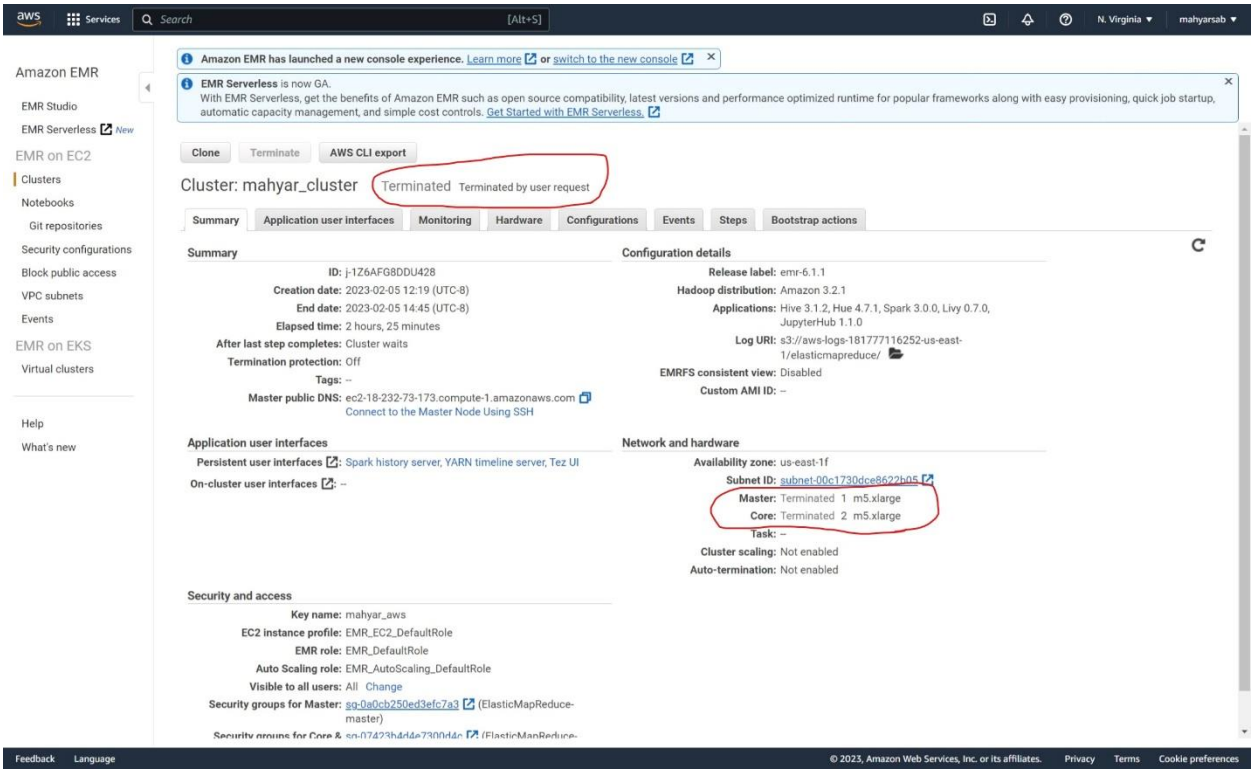
- hadoop fs -getmerge is a Hadoop command-line tool that merges multiple files in HDFS (Hadoop Distributed File System) into a single file on the local file system. The merged file will contain the contents of all the input files concatenated together, with a newline character separating each file's contents.



- After that I am going to type exit in the terminal and terminate the EMR cluster because we don’t need it anymore. The filtered file is now on our S3 bucket (our scalable storage on the cloud) and we will continue our work on my local machine.

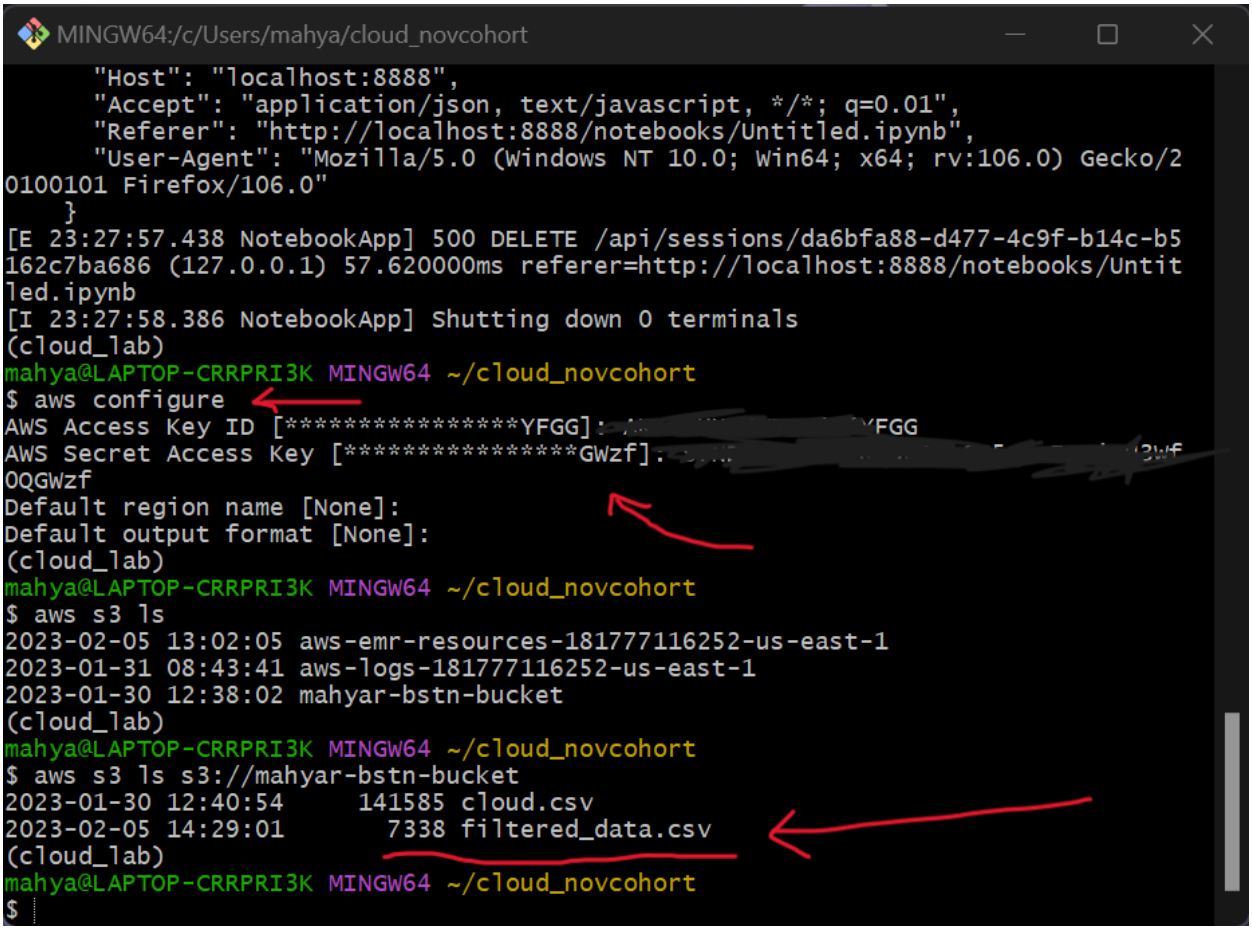

MINGW64:/c/Users/mahya/cloud_novcohort
data,1998,208230,104315,4231
data,1999,230573,108982,4179
data,2000,223813,107134,4040
data,2001,247209,115258,3983
data,2002,241848,109219,3825
data,2003,244250,105400,3684
data,2004,201841,93539,3483
data,2005,197467,88901,3392
data,2006,203669,92960,3449
data,2007,168338,78986,3246
data,2008,105331,47811,2358
[hadoop@ip-172-31-73-2 ~]$ hadoop fs -ls /user/hadoop/filtered_data
Found 3 items
-rw-r--r--   1 livy hadoop   0 2023-02-05 22:04 /user/hadoop/filtered_data/_SUCCESS
-rw-r--r--   1 livy hadoop   33 2023-02-05 22:03 /user/hadoop/filtered_data/part-00000-c5ac7f4e-c869-4f
f9-bd50-d96747ab572a-c000.csv
-rw-r--r--   1 livy hadoop   7305 2023-02-05 22:04 /user/hadoop/filtered_data/part-00023-c5ac7f4e-c869-4f
f9-bd50-d96747ab572a-c000.csv
[hadoop@ip-172-31-73-2 ~]$ hadoop fs -ls /user/hadoop/
Found 2 items
-rw-r--r--   1 hadoop hadoop 5292105197 2023-02-05 20:48 /user/hadoop/eng_1M_1gram
drwxr-xr-x   - livy   hadoop   0 2023-02-05 22:04 /user/hadoop/filtered_data
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$ hadoop fs -getmerge /user/hadoop/filtered_data filtered_data.csv
[hadoop@ip-172-31-73-2 ~]$ aws s3 cp filtered_data.csv s3://mahyar-bstn-bucket
upload: ./filtered_data.csv to s3://mahyar-bstn-bucket/filtered_data.csv
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$
[hadoop@ip-172-31-73-2 ~]$ exit
logout
Connection to ec2-18-232-73-173.compute-1.amazonaws.com closed.
(cloud_lab)
mahya@LAPTOP-CRRPRI3K MINGW64 ~/cloud_novcohort
$





- Now our cluster is successfully terminated and let's move to question 6:

Question 6: In this step, on my local machine (or on AWS outside of Spark) in python, I am going to read the CSV data from the S3 folder into a Pandas data frame. But before that, I must have first authenticated on my machine using AWS configure on the command line as below:



- Then I ran some codes to check whether the filtered file exists on my S3 bucket. Now we will open the jupyter note book and continue answering questions 6,7,8 there on local computer:

Remaining Questions:

6. On your local machine (or on AWS outside of Spark) in python, read the CSV data from the S3 folder into a pandas DataFrame (You will have to research how to read data into pandas from S3 buckets). Note You must have first authenticated on your machine using aws configure on the command line to complete this step).
7. Plot the number of occurrences of the token (the frequency column) of data over the years using matplotlib.
8. Compare Hadoop and Spark as distributed file systems:

   a. What are the advantages/ differences between Hadoop and Spark? List two advantages for each.

   b. Explain how the HDFS stores the data.

6

- In this step we should first authenticated on our machine using aws configure on the command line.
- Then we will use boto to read the csv data into pandas dataframe:

Using Boto3 to communicate with S3:

- First we will import packeges we need:

In [1]:
```
# Import packages

import boto3
import pandas as pd
import matplotlib.pyplot as plt
```


- First we will import packeges we need:

In [1]:
```
# Import packages

import boto3
import pandas as pd
import matplotlib.pyplot as plt
```


Let's start with the Client method. We will create a boto3 client object and pass it the AWS service we would like to use, which in this case is S3.

In [2]:
```
# Instantiate an S3 client
s3_client = boto3.client('s3')

# Check
s3_client
```

Out[2]:
```
<botocore.client.S3 at 0x2632244ebb0>
```

- Recall that this instance is linked to the AWS account authorized on your computer, and if you run this notebook you will see different outputs than what I have here.

In [3]:
```
# Send a request to list all my buckets, and assign the response to a variable
response = s3_client.list_buckets()

# Check
response
```

Out[3]:
```
{'ResponseMetadata': {'RequestId': 'MQBECBEXBA36ZK0R',
  'HostId': '0NY45PAlsx/U0lT5C727v7nRIPn60OFryeyfRPjXjD0/wID9q88j6/tsM0VeDEPmvE/A887dvhM=',
  'HTTPStatusCode': 200,
  'HTTPHeaders': {'x-amz-id-2': '0NY45PAlsx/U0lT5C727v7nRIPn60OFryeyfRPjXjD0/wID9q88j6/tsM0VeDEPmvE/A887dvhM=',
   'x-amz-request-id': 'MQBECBEXBA36ZK0R',
   'date': 'Mon, 06 Feb 2023 10:05:25 GMT',
   'content-type': 'application/xml',
   'transfer-encoding': 'chunked',
   'server': 'AmazonS3'},
  'RetryAttempts': 0},
 'Buckets': [{'Name': 'aws-emr-resources-181777116252-us-east-1',
```

```
 'RetryAttempts': 0},
'Buckets': [{'Name': 'aws-emr-resources-181777116252-us-east-1',
  'CreationDate': datetime.datetime(2023, 2, 5, 21, 2, 5, tzinfo=tzutc())},
 {'Name': 'aws-logs-181777116252-us-east-1',
  'CreationDate': datetime.datetime(2023, 1, 31, 16, 43, 41, tzinfo=tzutc())},
 {'Name': 'mahyar-bstn-bucket',
  'CreationDate': datetime.datetime(2023, 1, 30, 20, 38, 2, tzinfo=tzutc())}],
'Owner': {'DisplayName': 'mahyar_sabooni',
 'ID': '7eb122d16e9fa085284e3de4a147f08b6b4a882422396d0227276f95079a5618'}}
```

- We can access the keys and values of a dictionary using the .keys() and .values() methods, respectively:

In [4]:
```
# The keys

response.keys()
```

Out[4]:
```
dict_keys(['ResponseMetadata', 'Buckets', 'Owner'])
```

In [5]:
```
# The values

response.values()
```

Out[5]:
```
dict_values([{'RequestId': 'MQBECBEXBA36ZK0R', 'HostId': '0NY45PAlsx/U0lT5C727v7nRIPn60OFryeyfRPjXjD0/wID9q88j6/tsM0VeDEPmvE
/A887dvhM=', 'HTTPStatusCode': 200, 'HTTPHeaders': {'x-amz-id-2': '0NY45PAlsx/U0lT5C727v7nRIPn60OFryeyfRPjXjD0/wID9q88j6/tsM
0VeDEPmvE/A887dvhM=', 'x-amz-request-id': 'MQBECBEXBA36ZK0R', 'date': 'Mon, 06 Feb 2023 10:05:25 GMT', 'content-type': 'appl
ication/xml', 'transfer-encoding': 'chunked', 'server': 'AmazonS3'}, 'RetryAttempts': 0}, [{'Name': 'aws-emr-resources-18177
7116252-us-east-1', 'CreationDate': datetime.datetime(2023, 2, 5, 21, 2, 5, tzinfo=tzutc())}, {'Name': 'aws-logs-18177711625
2-us-east-1', 'CreationDate': datetime.datetime(2023, 1, 31, 16, 43, 41, tzinfo=tzutc())}, {'Name': 'mahyar-bstn-bucket', 'C
reationDate': datetime.datetime(2023, 1, 30, 20, 38, 2, tzinfo=tzutc())}], {'DisplayName': 'mahyar_sabooni', 'ID': '7eb122d1
6e9fa085284e3de4a147f08b6b4a882422396d0227276f95079a5618'}])
```

- The values aren't easy for us to work with, so let's use the keys instead:

In [6]:
```
# What's the output if I index the big dictionary by the 'Buckets' key?

response['Buckets']
```

In [6]:
```
# What's the output if I index the big dictionary by the 'Buckets' key?

response['Buckets']
```

Out[6]:
```
[{'Name': 'aws-emr-resources-181777116252-us-east-1',
  'CreationDate': datetime.datetime(2023, 2, 5, 21, 2, 5, tzinfo=tzutc())},
 {'Name': 'aws-logs-181777116252-us-east-1',
  'CreationDate': datetime.datetime(2023, 1, 31, 16, 43, 41, tzinfo=tzutc())},
 {'Name': 'mahyar-bstn-bucket',
  'CreationDate': datetime.datetime(2023, 1, 30, 20, 38, 2, tzinfo=tzutc())}]
```

In [7]:
```
# Type of the output above

type(response['Buckets'])
```

Out[7]:
```
list
```

- Now let's print the names of my buckets:

In [9]:
```
# Instantiate a resource service client

s3_resource = boto3.resource('s3')

# Iterate over the buckets

for bucket in s3_resource.buckets.all():

    # Use the `.name` method on each bucket to print its name
    print(bucket.name)
```

```
aws-emr-resources-181777116252-us-east-1
aws-logs-181777116252-us-east-1
mahyar-bstn-bucket
```

- Accessing and reading in a specific file:

- Accessing and reading in a specific file:

In [10]:
```
# Get the specified file from the specified bucket and assign it to a variable

s3_object = s3_client.get_object(Bucket='mahyar-bstn-bucket', Key='filtered_data.csv')

# Check

s3_object
```

Out[10]:
```
{'ResponseMetadata': {'RequestId': 'QQ1Z87ZJA41N3FEX',
  'HostId': 'pFXqElHUzy1FbNP4PW8ufEFAqh3/JF24yOfyG6LiC7YTQ7ke3Z/KY+9PGoRl5aGp1OmTG4LytF8=',
  'HTTPStatusCode': 200,
  'HTTPHeaders': {'x-amz-id-2': 'pFXqElHUzy1FbNP4PW8ufEFAqh3/JF24yOfyG6LiC7YTQ7ke3Z/KY+9PGoRl5aGp1OmTG4LytF8=',
   'x-amz-request-id': 'QQ1Z87ZJA41N3FEX',
   'date': 'Mon, 06 Feb 2023 10:11:14 GMT',
   'last-modified': 'Sun, 05 Feb 2023 22:29:01 GMT',
   'etag': '"fb3cf3d45b1fc32b939cbd13697dbe68"',
   'x-amz-server-side-encryption': 'AES256',
   'accept-ranges': 'bytes',
   'content-type': 'text/csv',
   'server': 'AmazonS3',
   'content-length': '7338'},
  'RetryAttempts': 0},
 'AcceptRanges': 'bytes',
 'LastModified': datetime.datetime(2023, 2, 5, 22, 29, 1, tzinfo=tzutc()),
 'ContentLength': 7338,
 'ETag': '"fb3cf3d45b1fc32b939cbd13697dbe68"',
 'ContentType': 'text/csv',
 'ServerSideEncryption': 'AES256',
 'Metadata': {},
 'Body': <botocore.response.StreamingBody at 0x26322550820>}
```

In [11]:
```
# Index the above by the 'Body' key

s3_object['Body']
```

Out[11]:
```
<botocore.response.StreamingBody at 0x26322550820>
```

- Now let's read the data:

In [12]:
```
# Read the csv into a dataframe; note that we pass in the 'StreamingBody' above

s3_filtered_data = pd.read_csv(s3_object['Body'] , header = 1)

# Check

s3_filtered_data
```

Out[12]:

	token	year	frequency	pages	books
0	data	1584	16	14	1
1	data	1614	3	2	1
2	data	1627	1	1	1
3	data	1631	22	18	1
4	data	1637	1	1	1
...	...	...	...	...	...
311	data	2004	201841	93539	3483
312	data	2005	197467	88901	3392
313	data	2006	203669	92960	3449
314	data	2007	168338	78986	3246
315	data	2008	105331	47811	2358

316 rows × 5 columns

- **This is one way to read the data with boto3. we can even read the data directly into Pandas with the code below:**

Reading data directly into pandas from s3 bucket:

In [13]:
```
# Read the csv from the specified S3 path into a dataframe

filtered_data_directly_pandas = pd.read_csv('s3://mahyar-bstn-bucket/filtered_data.csv', header = 1 )

# Check

filtered_data_directly_pandas
```

Out[13]:

	token	year	frequency	pages	books
0	data	1584	16	14	1
1	data	1614	3	2	1
2	data	1627	1	1	1
3	data	1631	22	18	1
4	data	1637	1	1	1
...	...	...	...	...	...
311	data	2004	201841	93539	3483
312	data	2005	197467	88901	3392
313	data	2006	203669	92960	3449
314	data	2007	168338	78986	3246
315	data	2008	105331	47811	2358

316 rows × 5 columns

- Now let's check the shape, info and some basic math calculations of the numeric columns:

In [14]:
```
filtered_data_directly_pandas.describe()
```

Out[14]:

	year	frequency	pages	books
count	316.000000	316.000000	316.000000	316.000000
mean	1847.569620	38555.993671	21711.041139	1493.110759

- Now let's check the shape, info and some basic math calculations of the numeric columns:

In [14]:
```
filtered_data_directly_pandas.describe()
```


Out[14]:

	year	frequency	pages	books
count	316.000000	316.000000	316.000000	316.000000
mean	1847.569620	38555.993671	21711.041139	1493.110759
std	96.874382	69212.366418	34901.797740	1560.040802
min	1584.000000	1.000000	1.000000	1.000000
25%	1771.750000	20.750000	19.000000	11.750000
50%	1850.500000	3004.000000	2729.500000	868.000000
75%	1929.250000	41776.750000	30195.750000	2964.250000
max	2008.000000	254561.000000	122472.000000	4372.000000

In [15]:
```
filtered_data_directly_pandas.shape
```

Out[15]: (316, 5)

- Our data has 316 rows and 5 columns.

In [16]:
```
filtered_data_directly_pandas.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 316 entries, 0 to 315
Data columns (total 5 columns):
 #   Column     Non-Null Count  Dtype 
---  ------     --------------  ----- 
 0   token      316 non-null    object
 1   year       316 non-null    int64 
 2   frequency  316 non-null    int64 
 3   pages      316 non-null    int64 
 4   books      316 non-null    int64 
dtypes: int64(4), object(1)
memory usage: 12.5+ KB
```

Question 7: I will plot the frequency over year using Matplotlib:

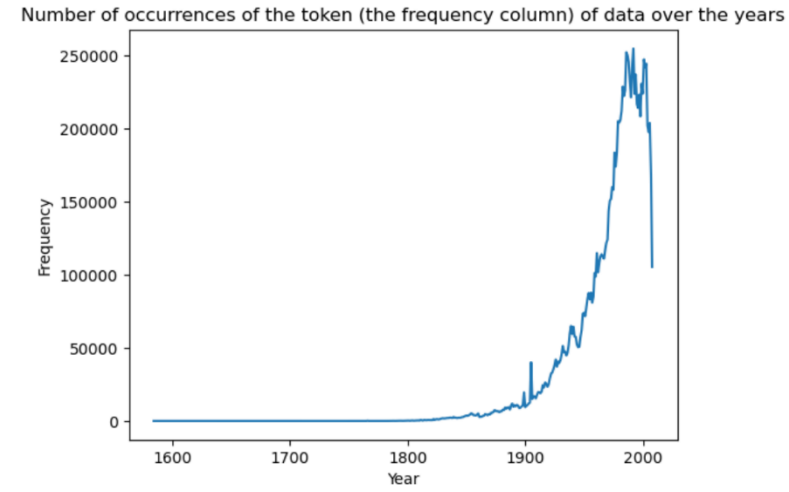
- We have 4 nymeric columns and one object column(token). We have no null values.
- **In this question we read the data into a pandas dataframe in 2 ways and then we did some basic describe on the dataset.**

7

- Now we will plot the number of occurrences of the token (the frequency column) of data over the years using matplotlib:

In [18]:

```
import matplotlib.pyplot as plt

# Group the data by year and calculate the sum of the frequency column

grouped_data = filtered_data_directly_pandas.groupby("year").sum()

# Plot the sum of the frequency column for each year

plt.plot(grouped_data.index, grouped_data["frequency"])

# Add labels to the x and y axes

plt.title("Number of occurrences of the token (the frequency column) of data over the years")
plt.xlabel("Year")
plt.ylabel("Frequency")

# Show the plot

plt.show()

```



- Question 8: Compare Hadoop and Spark as distributed file systems.

- A. What are the advantages/ differences between Hadoop and Spark? List two advantages for each.

8.a : What are the advantages/ differences between Hadoop and Spark? List two advantages for each.

Hadoop and Spark are both distributed file systems, but they have some differences in terms of their design and functionality:

Advantages of Hadoop:

1. Scalability: Hadoop can scale to handle very large data sets by adding more nodes to the cluster.
2. Cost-effective: Hadoop is an open-source solution, so there are no licensing costs involved, and it can run on commodity hardware.

Advantages of Spark:

1. Speed: Spark is much faster than Hadoop MapReduce for big data processing because it uses in-memory computing and can cache intermediate data in memory for faster access.
2. Ease of use: Spark has a high-level API that makes it easier for developers to write and maintain big data processing code, compared to Hadoop MapReduce, which has a low-level API.

It is worth noting that Hadoop and Spark are complementary technologies, and Spark can be used on top of Hadoop as an alternative to MapReduce for big data processing.

- B. Explain how the HDFS stores the data.

8.b : Explain how the HDFS stores the data.

HDFS (Hadoop Distributed File System) is the primary storage system used by Hadoop. HDFS stores data as large files that are broken down into smaller blocks and distributed across multiple nodes in a Hadoop cluster. The blocks are replicated across multiple nodes in the cluster for increased reliability and to provide fault tolerance in case a node fails.

When a file is written to HDFS, it is divided into blocks and each block is stored on multiple nodes. The blocks are stored on the nodes close to the data, reducing network latency and increasing data transfer speeds. The blocks are also replicated to other nodes in the cluster, providing backup copies of the data in case a node fails.

HDFS uses a master/slave architecture, with a NameNode as the master and DataNodes as the slaves. The NameNode maintains the file system namespace, manages the mapping of blocks to DataNodes, and coordinates access to files stored in HDFS. The DataNodes are responsible for storing the actual data blocks and serving read and write requests from clients.

In summary, HDFS is a highly scalable, fault-tolerant, and distributed file system that provides reliable data storage for Hadoop clusters.

- I attached the filtered_data.csv file to the project folder too in case you want to check it.

Thank You for reading my work!