EECS 6893
HW1
Yutao Zhou
UNI: yz4395

1. Iterative K-means clustering on Spark

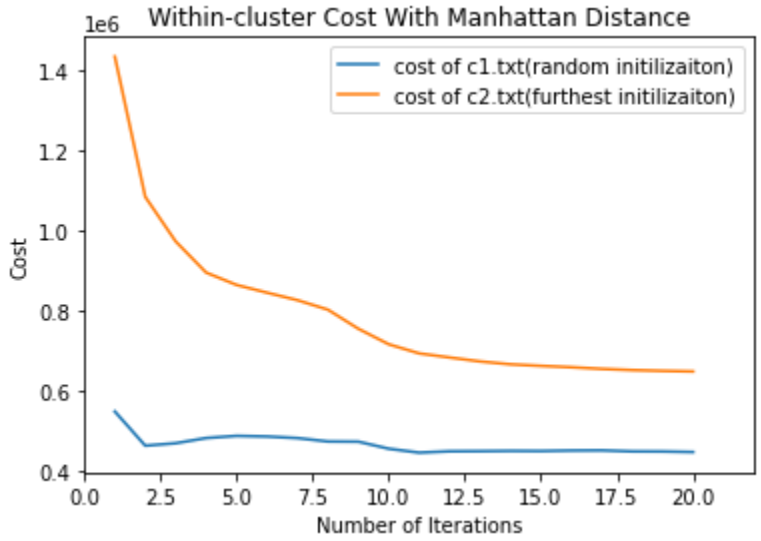
(1)

```python
import operator
import sys
from pyspark import SparkConf, SparkContext
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
from pyspark.mllib.feature import Normalizer
from sklearn.manifold import TSNE

# Macros.
MAX_ITER = 20
DATA_PATH = "gs://eecs6893_data/q1/data.txt"
C1_PATH = "gs://eecs6893_data/q1/c1.txt"
C2_PATH = "gs://eecs6893_data/q1/c2.txt"
NORM = 2

# Helper functions.
def closest(p, centroids, norm):
    """
    Compute closest centroid for a given point.
    Args:
        p (numpy.ndarray): input point
        centroids (list): A list of centroids points
        norm (int): 1 or 2
    Returns:
        int: The index of closest centroid.
    """
    closest_c = min([(i, linalg.norm(p - c, norm))
                     for i, c in enumerate(centroids)],
                     key=operator.itemgetter(1))[0]
    return closest_c

# K-means clustering
```

```
def getCloestCentroid(pt, centroids, norm=2):
    c_index = closest(pt, centroids, norm)
    cost = linalg.norm(pt - centroids[c_index], norm)
    return (c_index, (pt, 1, cost))

def calculateCost(costData):
    totalCost, numPoints = 0, 0
    for cluster in costData:
        totalCost += cluster[1][2]
        numPoints += cluster[1][1]
    return totalCost

def kmeans(data, centroids, norm=2):
    """
    Conduct k-means clustering given data and centroid.
    This is the basic version of k-means, you might need more
    code to record cluster assignment to plot TSNE, and more
    data structure to record cost.
    Args:
        data (RDD): RDD of points
        centroids (list): A list of centroids points
        norm (int): 1 or 2
    Returns:
        RDD: assignment information of points, a RDD of (centroid, (point, 1))
        list: a list of centroids
        and define yourself...
    """
    # iterative k-means
    cost = []
    for _ in range(MAX_ITER):
        # Transform each point to a combo of point, closest centroid, count=1
        # point -> (closest_centroid, (point, 1))
        clusters = data.map(lambda pt: getCloestCentroid(pt, centroids, norm)).reduceByKey(lambda count1, count2: \
```

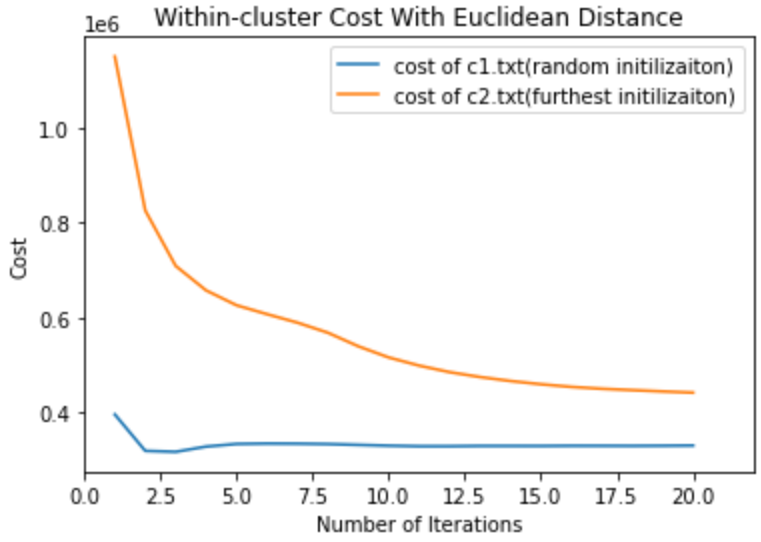
```
            (count1[0] + count2[0], count1[1] + count2[1], count1[2] + count2[2]))
        # Re-compute cluster center
        # For each cluster center (key), aggregate its values
        centroids = clusters.map(lambda x: x[1][0]/x[1][1]).collect()
        # by summing up points and count
        # Average the points for each centroid: divide sum of points by count
        # Use collect() to turn RDD into list
        cost.append(calculateCost(clusters.collect()))
    return cost, centroids

def plotCost(c1, c2, distanceType, MAX_ITER):
    fig,ax1 = plt.subplots()
    x = list(range(1, MAX_ITER + 1))
    c1l = ax1.plot(x,c1,'-', label = f"cost of c1.txt(random initilizaiton)")
    c2l = ax1.plot(x,c2,'-', label = f"cost of c2.txt(furthest initilizaiton)")
    ax1.set_xlim(0, MAX_ITER + 2)
    ax1.set_xlabel("Number of Iterations")
    ax1.set_ylabel("Cost")
    ax1.legend()
    ax1.set_title(f"Within-cluster Cost With {distanceType} Distance")
    fig.show()

def tSNE(data, centroids, norm, plotTitle):
    def findCluster(pt, centroids, norm=2):
        c_index = closest(pt, centroids, norm)
        return (c_index)
    def plotTSNE(embeddedData, plotTitle, c=None):
        x = embeddedData[:, 0]
        y = embeddedData[:, 1]
        plt.scatter(x, y, c=c, cmap=plt.cm.get_cmap("jet", 10))
        plt.title(plotTitle)
        plt.show()
    dataClusters = data.map(lambda pt: findCluster(pt, centroids, norm))
```

```
    cluster = dataClusters.collect()
    originalData = np.array(data.collect())
    orignalEmbeddedData = TSNE(n_components=2, random_state=100).fit_transform(originalData)
    plotTSNE(orignalEmbeddedData,plotTitle)
    plotTSNE(orignalEmbeddedData, plotTitle, cluster)

def main():
    # Spark settings
    conf = SparkConf()
    sc = SparkContext(conf=conf)

    # Load the data, cache this since we're accessing this each iteration
    data = sc.textFile(DATA_PATH).map(
            lambda line: np.array([float(x) for x in line.split(' ')])
            ).cache()
    # Load the initial centroids c1, split into a list of np arrays
    centroids1 = sc.textFile(C1_PATH).map(
            lambda line: np.array([float(x) for x in line.split(' ')])
            ).collect()
    # Load the initial centroids c2, split into a list of np arrays
    centroids2 = sc.textFile(C2_PATH).map(
            lambda line: np.array([float(x) for x in line.split(' ')])
            ).collect()

    # TODO: Run the kmeans clustering and complete the HW
    c1, centroids1Out = kmeans(data, centroids1, NORM)
    c2, centroids2Out = kmeans(data, centroids2, NORM)
    if NORM == 1:
        distanceType = "Manhattan"
    elif NORM == 2:
        distanceType = "Euclidean"
    plotCost(c1, c2, distanceType, MAX_ITER)
    # tSNE(data, centroids1Out, NORM, "t-SNE results of c1.txt using L2 distance and random_state=100")
    # tSNE(data, centroids2Out, NORM, "t-SNE results of c2.txt using L2 distance and random_state=100")

if __name__ == "__main__":
    main()
```



(2)

We simply change the NORM in part (1) to 2 and re-run the program. The output graph is as follows.


Within-cluster Cost With Euclidean Distance
1e6
cost of c1.txt(random initilizaiton)
cost of c2.txt(furthest initilizaiton)
1.0
0.8
0.6
0.4
Cost
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Number of Iterations

(3)

```python
import operator
import sys
from pyspark import SparkConf, SparkContext
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg
from pyspark.mllib.feature import Normalizer
from sklearn.manifold import TSNE

# Macros.
MAX_ITER = 20
DATA_PATH = "gs://eecs6893_data/q1/data.txt"
C1_PATH = "gs://eecs6893_data/q1/c1.txt"
C2_PATH = "gs://eecs6893_data/q1/c2.txt"
NORM = 2

# Helper functions.
def closest(p, centroids, norm):
    """
    Compute closest centroid for a given point.
    Args:
        p (numpy.ndarray): input point
        centroids (list): A list of centroids points
        norm (int): 1 or 2
    Returns:
        int: The index of closest centroid.
    """
    closest_c = min([(i, linalg.norm(p - c, norm))
                     for i, c in enumerate(centroids)],
                     key=operator.itemgetter(1))[0]
    return closest_c

# K-means clustering
```

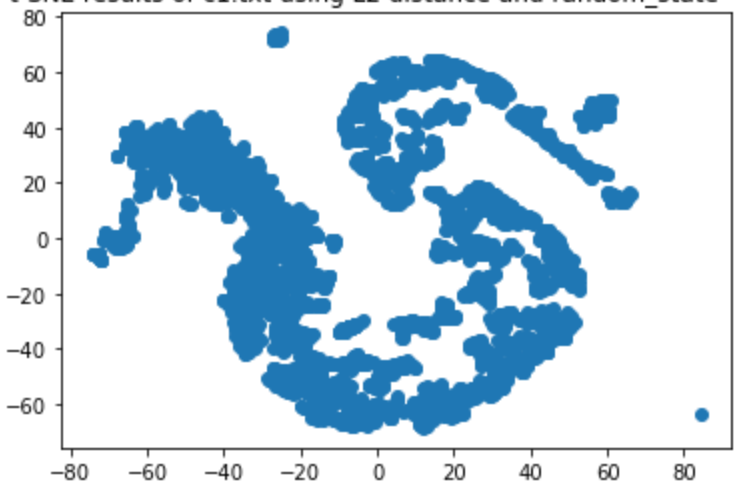
```python
def getCloestCentroid(pt, centroids, norm=2):
    c_index = closest(pt, centroids, norm)
    cost = linalg.norm(pt - centroids[c_index], norm)
    return (c_index, (pt, 1, cost))

def calculateCost(costData):
    totalCost, numPoints = 0, 0
    for cluster in costData:
        totalCost += cluster[1][2]
        numPoints += cluster[1][1]
    return totalCost

def kmeans(data, centroids, norm=2):
    """
    Conduct k-means clustering given data and centroid.
    This is the basic version of k-means, you might need more
    code to record cluster assignment to plot TSNE, and more
    data structure to record cost.
    Args:
        data (RDD): RDD of points
        centroids (list): A list of centroids points
        norm (int): 1 or 2
    Returns:
        RDD: assignment information of points, a RDD of (centroid, (point, 1))
        list: a list of centroids
        and define yourself...
    """
    # iterative k-means
    cost = []
    for _ in range(MAX_ITER):
        # Transform each point to a combo of point, closest centroid, count=1
        # point -> (closest_centroid, (point, 1))
        clusters = data.map(lambda pt: getCloestCentroid(pt, centroids, norm)).reduceByKey(lambda count1, count2: \
```

```
            (count1[0] + count2[0], count1[1] + count2[1], count1[2] + count2[2]))
        # Re-compute cluster center
        # For each cluster center (key), aggregate its values
        centroids = clusters.map(lambda x: x[1][0]/x[1][1]).collect()
        # by summing up points and count
        # Average the points for each centroid: divide sum of points by count
        # Use collect() to turn RDD into list
        cost.append(calculateCost(clusters.collect()))
    return cost, centroids

def plotCost(c1, c2, distanceType, MAX_ITER):
    fig,ax1 = plt.subplots()
    x = list(range(1, MAX_ITER + 1))
    c1l = ax1.plot(x,c1,'-', label = f"cost of c1.txt(random initilizaiton)")
    c2l = ax1.plot(x,c2,'-', label = f"cost of c2.txt(furthest initilizaiton)")
    ax1.set_xlim(0, MAX_ITER + 2)
    ax1.set_xlabel("Number of Iterations")
    ax1.set_ylabel("Cost")
    ax1.legend()
    ax1.set_title(f"Within-cluster Cost With {distanceType} Distance")
    fig.show()

def tSNE(data, centroids, norm, plotTitle):
    def findCluster(pt, centroids, norm=2):
        c_index = closest(pt, centroids, norm)
        return (c_index)
    def plotTSNE(embeddedData, plotTitle, c=None):
        x = embeddedData[:, 0]
        y = embeddedData[:, 1]
        plt.scatter(x, y, c=c, cmap=plt.cm.get_cmap("jet", 10))
        plt.title(plotTitle)
        plt.show()
    dataClusters = data.map(lambda pt: findCluster(pt, centroids, norm))
```

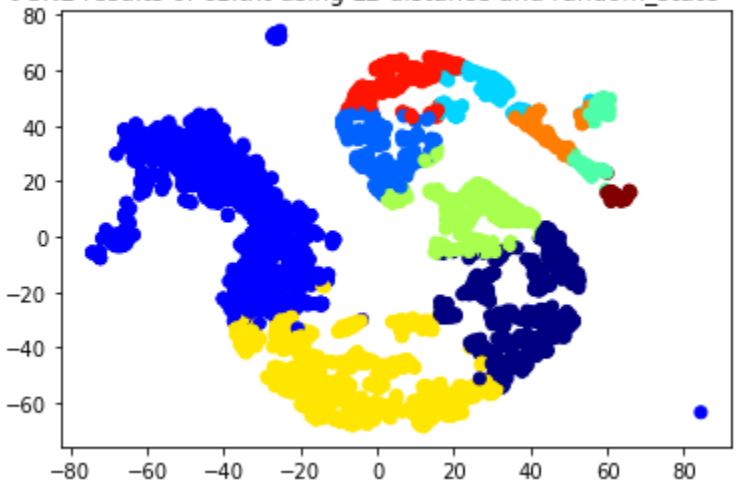
```
    cluster = dataClusters.collect()
    originalData = np.array(data.collect())
    orignalEmbeddedData = TSNE(n_components=2, random_state=100).fit_transform(originalData)
    plotTSNE(orignalEmbeddedData,plotTitle)
    plotTSNE(orignalEmbeddedData, plotTitle, cluster)

def main():
    # Spark settings
    conf = SparkConf()
    sc = SparkContext(conf=conf)

    # Load the data, cache this since we're accessing this each iteration
    data = sc.textFile(DATA_PATH).map(
            lambda line: np.array([float(x) for x in line.split(' ')])
            ).cache()
    # Load the initial centroids c1, split into a list of np arrays
    centroids1 = sc.textFile(C1_PATH).map(
            lambda line: np.array([float(x) for x in line.split(' ')])
            ).collect()
    # Load the initial centroids c2, split into a list of np arrays
    centroids2 = sc.textFile(C2_PATH).map(
            lambda line: np.array([float(x) for x in line.split(' ')])
            ).collect()

    # TODO: Run the kmeans clustering and complete the HW
    c1, centroids1Out = kmeans(data, centroids1, NORM)
    c2, centroids2Out = kmeans(data, centroids2, NORM)
    if NORM == 1:
        distanceType = "Manhattan"
    elif NORM == 2:
        distanceType = "Euclidean"
    # plotCost(c1, c2, distanceType, MAX_ITER)
    tSNE(data, centroids1Out, NORM, "t-SNE results of c1.txt using L2 distance and random_state=100")
```

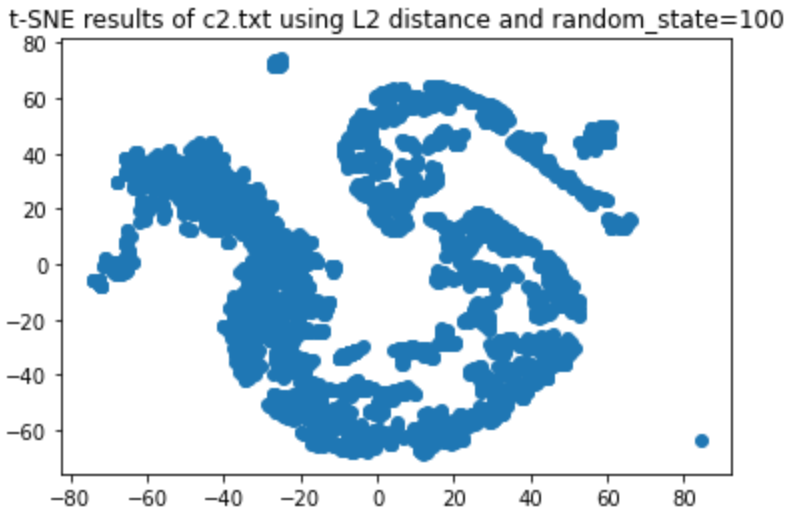
```
    tSNE(data, centroids2Out, NORM, "t-SNE results of c2.txt using L2 distance and random_state=100")

if __name__ == "__main__":
    main()
```

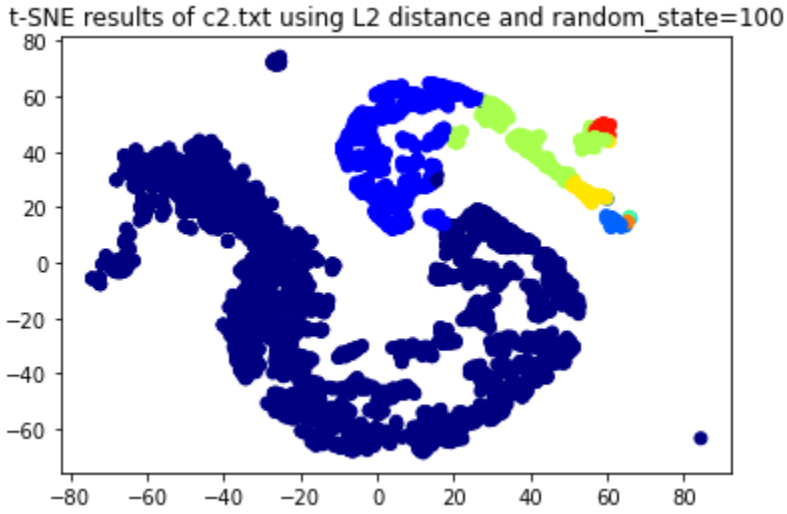
t-SNE results of c1.txt using L2 distance and random_state=100



t-SNE results of c1.txt using L2 distance and random_state=100







(4)

For L2 and L1 distance, random initialization of K-means using c1.txt is better than initialization using c2.txt in terms of cost. As we can see from my plot before. The initial and final within-closter cost of c1.txt is better than c2.txt. This make sense because if we initilize cluster centers to be as far away from each other as possible then all points would be far away from these points. As a result it have a higher initial cost. The cost keep decreasing in those 20 interactions. However, it is flatten out and the cost is not decreasing. The final cost for c2.txt is still higer than c1.txt. c2.txt is trying to have 10 independent clusters that are not far from boundary in this data set. However, c1.txt is more like grouping them together in relatively dense area. The cost of c1.txt is not decreasing much because there are simply too much data point around the center of those clusters and is it hard for the clusters to move to a beeter center. All in all, the cost of c1.txt is still lowever than c2.txt at all time.

(5)

The time complexity of the iterative K-means is O(number of points * number of centroids * number of iterations)

2. Monitoring Hadoop metrics

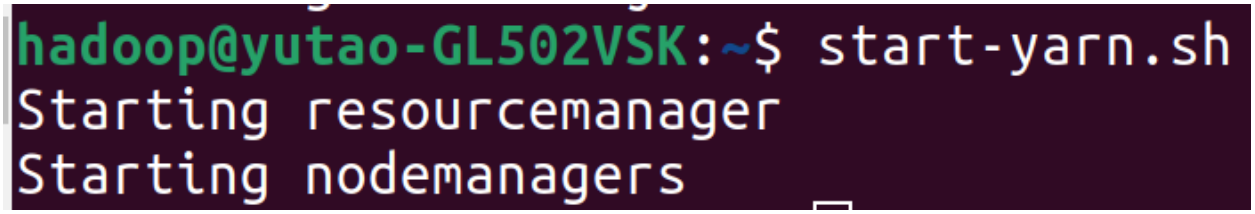
(1)

```
$ ssh localhost
hadoop@localhost's password:
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-48-generic x86_64)

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

0 updates can be applied immediately.

Last login: Mon Oct  3 10:51:53 2022 from 127.0.0.1
```

```
hadoop@yutao-GL502VSK:/usr/local$ hadoop version
Hadoop 3.3.1
Source code repository https://github.com/apache/hadoop.git -r a3b9c37a397ad4188041dd80621bdeefc46885f2
Compiled by ubuntu on 2021-06-15T05:13Z
Compiled with protoc 3.7.1
From source with checksum 88a4ddb2299aca054416d6b7f81ca55
This command was run using /usr/local/hadoop/share/hadoop/common/hadoop-common-3.3.1.jar
```

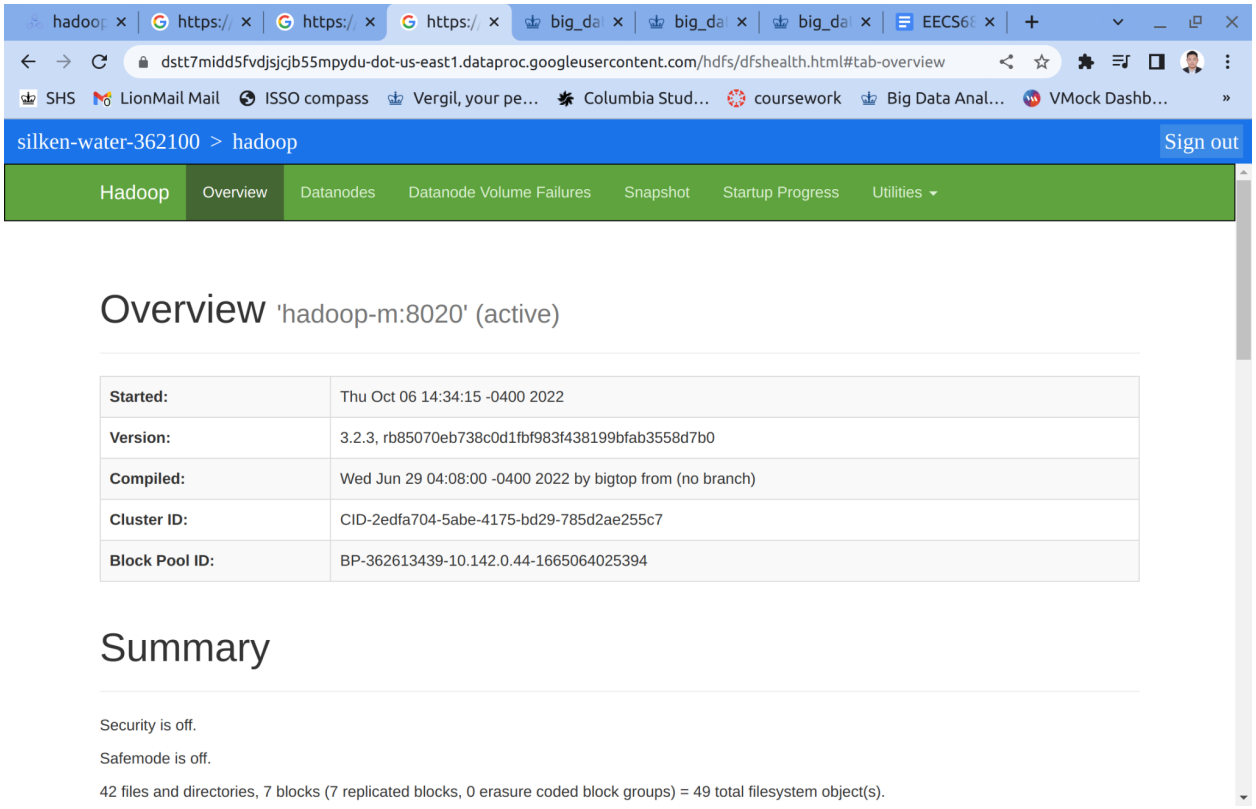
```
2022-10-03 11:51:52,247 INFO util.GSet: capacity      = 2^17 = 131072 entries
Re-format filesystem in Storage Directory root= /home/hadoop/hadoopinfra/hdfs/namenode; location= null ?
(Y or N) y
2022-10-03 11:51:59,985 INFO namenode.FSImage: Allocated new BlockPoolId: BP-832349331-127.0.1.1-16648123
19979
2022-10-03 11:51:59,985 INFO common.Storage: Will remove files: [/home/hadoop/hadoopinfra/hdfs/namenode/c
urrent/fsimage_0000000000000000000, /home/hadoop/hadoopinfra/hdfs/namenode/current/edits_0000000000000000
001-0000000000000000002, /home/hadoop/hadoopinfra/hdfs/namenode/current/edits_0000000000000000003-0000000
000000000003, /home/hadoop/hadoopinfra/hdfs/namenode/current/edits_inprogress_0000000000000000006, /home/
hadoop/hadoopinfra/hdfs/namenode/current/VERSION, /home/hadoop/hadoopinfra/hdfs/namenode/current/seen_txi
d, /home/hadoop/hadoopinfra/hdfs/namenode/current/fsimage_0000000000000000000.md5, /home/hadoop/hadoopinf
ra/hdfs/namenode/current/edits_0000000000000000004-0000000000000000005]
2022-10-03 11:51:59,996 INFO common.Storage: Storage directory /home/hadoop/hadoopinfra/hdfs/namenode has
 been successfully formatted.
2022-10-03 11:52:00,018 INFO namenode.FSImageFormatProtobuf: Saving image file /home/hadoop/hadoopinfra/h
dfs/namenode/current/fsimage.ckpt_0000000000000000000 using no compression
2022-10-03 11:52:00,095 INFO namenode.FSImageFormatProtobuf: Image file /home/hadoop/hadoopinfra/hdfs/nam
enode/current/fsimage.ckpt_0000000000000000000 of size 401 bytes saved in 0 seconds .
2022-10-03 11:52:00,102 INFO namenode.NNStorageRetentionManager: Going to retain 1 images with txid >= 0
2022-10-03 11:52:00,126 INFO namenode.FSNamesystem: Stopping services started for active state
2022-10-03 11:52:00,126 INFO namenode.FSNamesystem: Stopping services started for standby state
2022-10-03 11:52:00,129 INFO namenode.FSImage: FSImageSaver clean checkpoint: txid=0 when meet shutdown.
2022-10-03 11:52:00,129 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at yutao-GL502VSK/127.0.1.1
************************************************************/
```

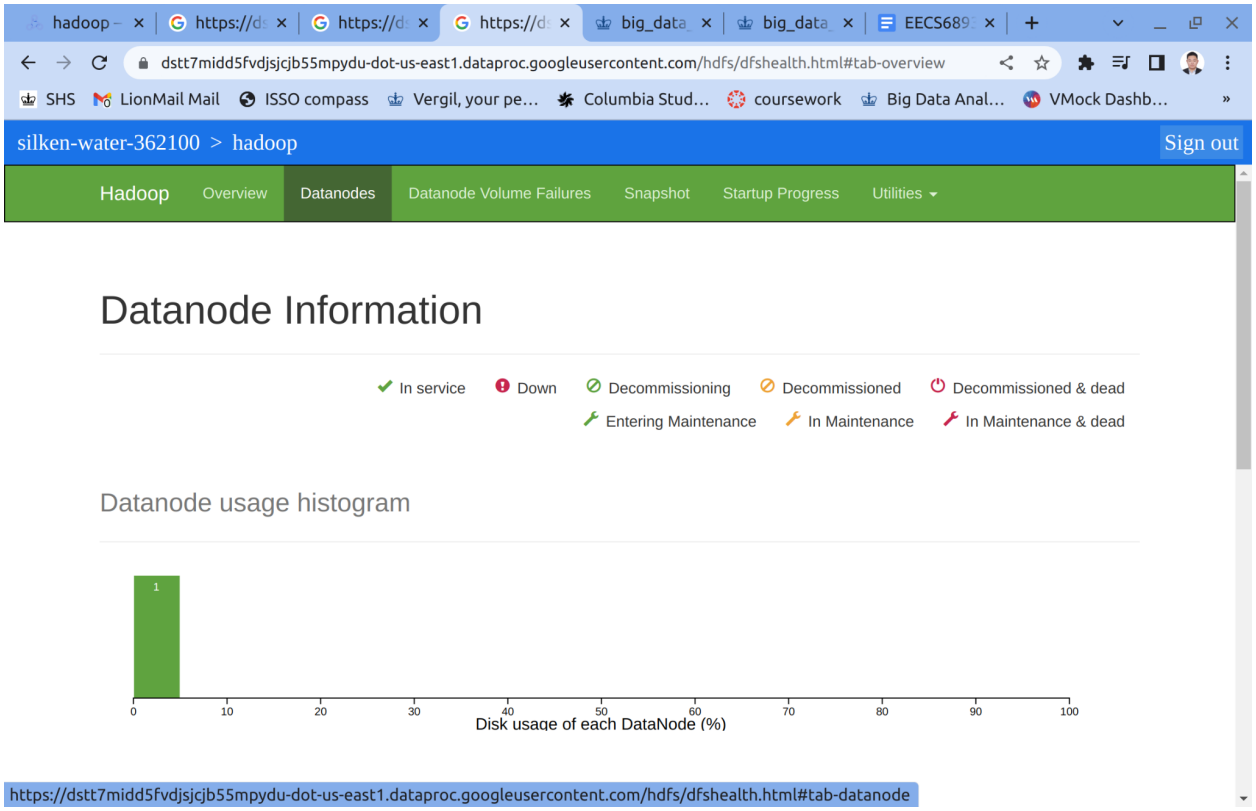
```
hadoop@yutao-GL502VSK:~$ start-dfs.sh
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [yutao-GL502VSK]
```

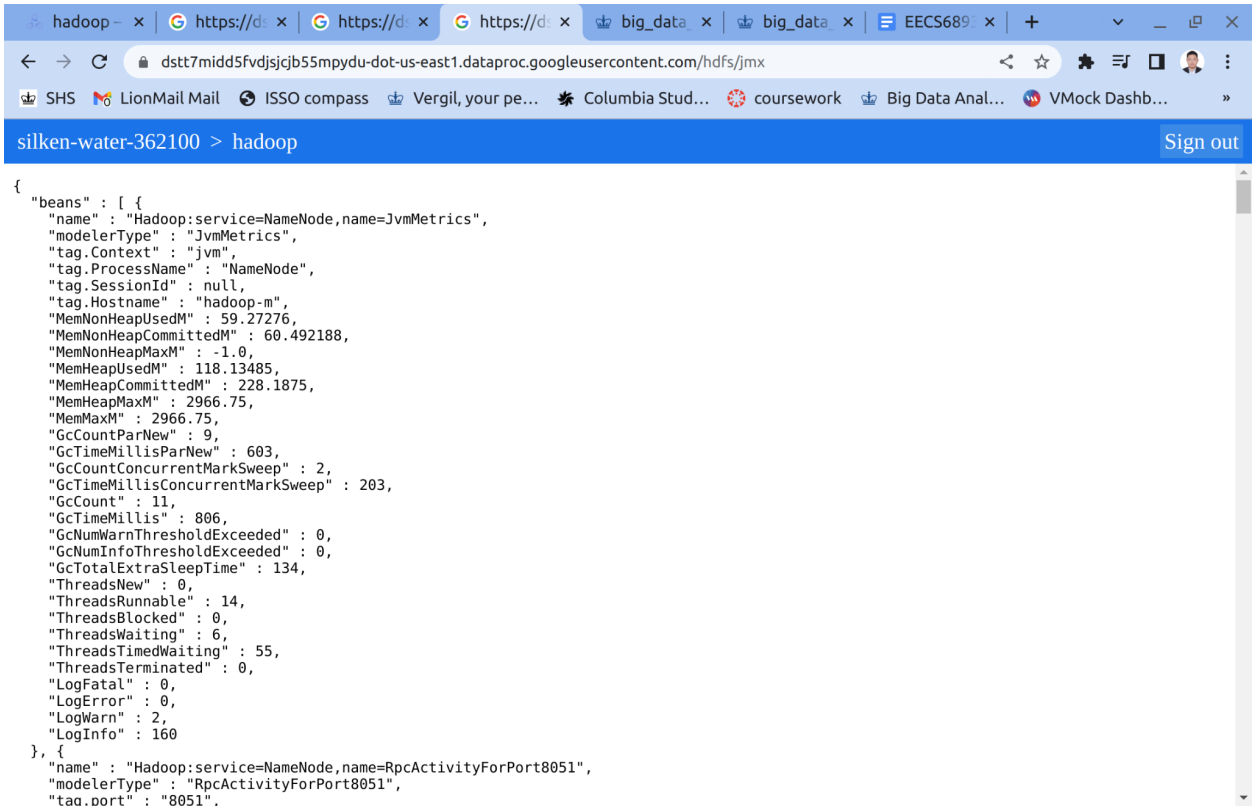


(2) HDFS metrics monitoring:

I chose to use Hadoop on GCP in the future because my windows keep messing up with my ubuntu somehow. Therefore, all of the following screenshots are from GCP.


dstt7midd5fvdjsjcjb55mpydu-dot-us-east1.dataproc.googleusercontent.com/hdfs/dfshealth.html#tab-overview
SHS
LionMail Mail
ISSO compass
Vergil, your pe…
Columbia Stud…
coursework
Big Data Anal…
VMock Dashb…
silken-water-362100 > hadoop
Sign out
Hadoop
Overview
Datanodes
Datanode Volume Failures
Snapshot
Startup Progress
Utilities
Overview 'hadoop-m:8020' (active)
Started: Thu Oct 06 14:34:15 -0400 2022
Version: 3.2.3, rb85070eb738c0d1fbf983f438199bfab3558d7b0
Compiled: Wed Jun 29 04:08:00 -0400 2022 by bigtop from (no branch)
Cluster ID: CID-2edfa704-5abe-4175-bd29-785d2ae255c7
Block Pool ID: BP-362613439-10.142.0.44-1665064025394
Summary
Security is off.
Safemode is off.
42 files and directories, 7 blocks (7 replicated blocks, 0 erasure coded block groups) = 49 total filesystem object(s).


dstt7midd5fvdjsjcjb55mpydu-dot-us-east1.dataproc.googleusercontent.com/hdfs/dfshealth.html#tab-overview
SHS
LionMail Mail
ISSO compass
Vergil, your pe…
Columbia Stud…
coursework
Big Data Anal…
VMock Dashb…
silken-water-362100 > hadoop
Sign out
Hadoop
Overview
Datanodes
Datanode Volume Failures
Snapshot
Startup Progress
Utilities
Datanode Information
In service
Down
Decommissioning
Decommissioned
Decommissioned & dead
Entering Maintenance
In Maintenance
In Maintenance & dead
Datanode usage histogram
1
0
10
20
30
40
50
60
70
80
90
100
Disk usage of each DataNode (%)



Five most important metrics:

1. StartupProgress

   This metric is used to show startup statistics for NameNode. Some people might say this is not important. But, I believe this is a metric that I will first check every time after I start the instance. This metric is important because we have to make sure the instance had started completely and properly then we could do calculations. Without checking this debugging might be painful. I will always make sure the PercentComplete is 1.0.

2. JvmMetrics

   This is a java vertical machine metric that is used to monitor system resources. This is important because it could let us know how much resource is our job using and we could infer what is the job status from this usage information(e.g. If it is not using resources it probably did not start running correctly).

3. RetryCache.NameNodeRetryCache

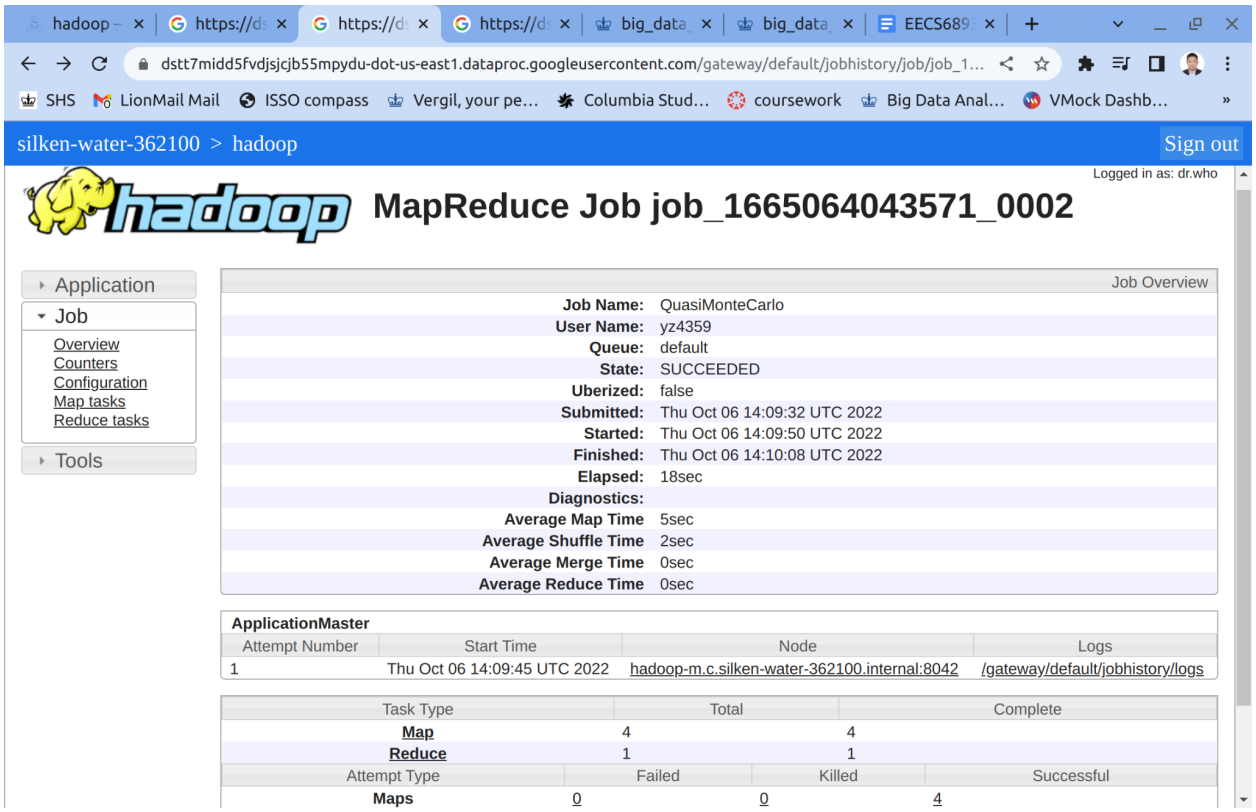
   This is a metric that is used to monitor the fail-over of NameNode. This is important because it can let us know how many times NameNode had NameNode hit RetryCashe to check for failure.

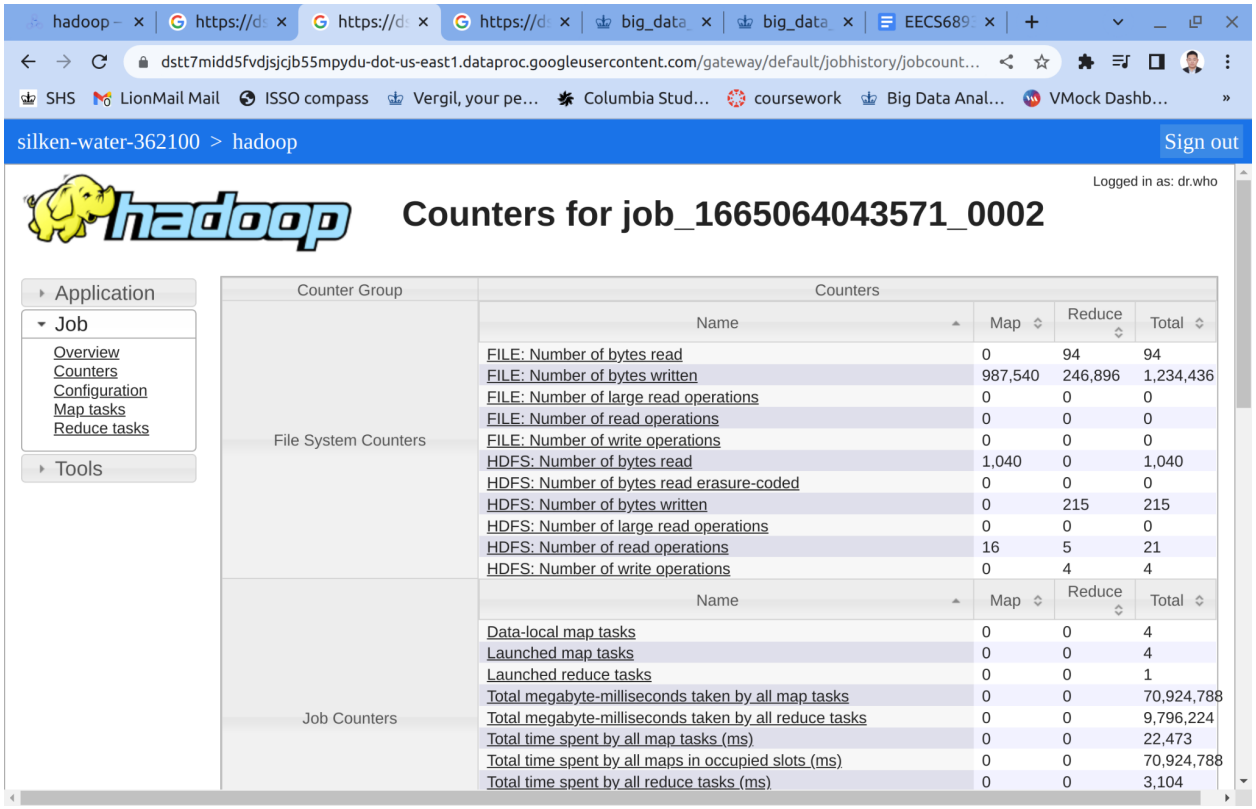
4. FSNamesystem

   This is a metric that is used to check some of the system's status for NameNode. This is important because it could give us some statistics on the usage of the system like disk usage which can be handy if we want to know whether we need to expand our hard disk.

5. rpc
   These metrics record rpc statistics like the number of bytes sent or received. This is important because we could see the total number of RPC calls and authentication or authorization failures. Therefore, we could see the healthiness of RPC.

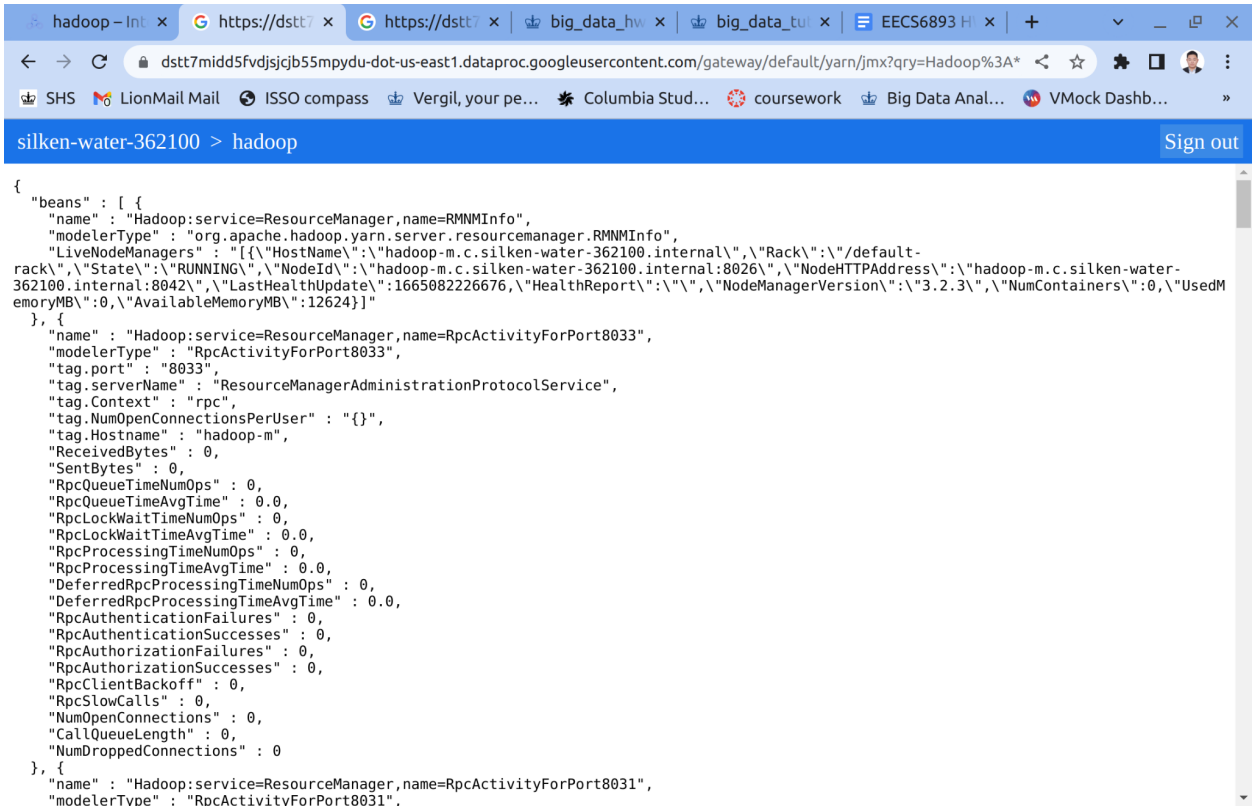
(3) MapReduce counters monitoring:





I could monitor the execution of map-reduced tasks by looking at job counters. The “Launched map tasks” will increment when it starts to map and the “Launched reduce tasks” will increment when it starts to reduce. Also, the respective totals spend for maps or reduce tasks will increment. Also, we could check the “Map-reduce framework” for CPU time spent for map and reduce respectively.

(4) YARN metrics monitoring:



Five most important metrics:

1. QueueMetrics

   This metric allows us to see the application queue from the resources manager's perspective. This is important because it could show us how many applications have been submitted and are running. Also, it would show come current resource usage. In other words, this will allow us to know where we are in the job queue in detail (how many jobs are running, pending, failed, killed, finished, etc).

2. JvmMetrics

   This is a java vertical machine matric that is used to monitor system resources. This is important because it could let us know how much resource is our job using and we could infer what is the job status from this usage information(e.g. If it is not using resources it probably did not start running correctly).

3. ClusterMetrics

   This metric is used to show the YARN cluster's metric. This is important because it could give us information about NodeManagers' status. We could know the system status in terms of nodes (e.g. number of NodeManager that are active, unhealthy, lost, etc).

4. NodeManagerMetrics

This metric is used to show general statistics for containers in general in the node. This is important because it could give us information about how many containers are there in that node, and how are they doing (e.g. whether they are running, etc).

5. ContainerMetrics

   This metric is used to show resource statistics for a specific container in the node. This is important because it could give us information about one specific container's status. In other words, we zoomed in from NodeManager to a container.