Big Data Assignment: K-means Clustering and Hadoop MapReduce

Table of Contents

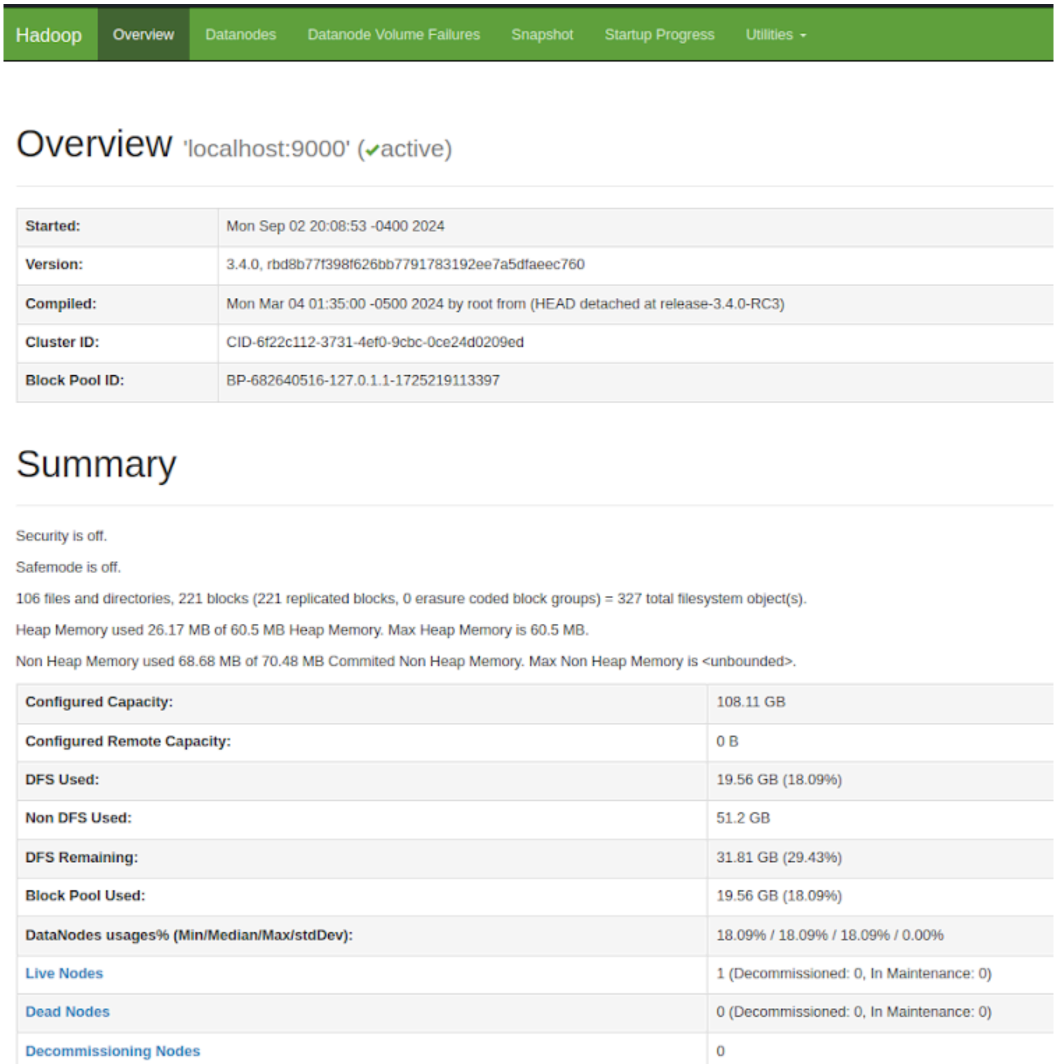
1. Introduction

This document presents the solution to the Big Data Assignment involving 2 components:

1. The implementation of the Word Count MapReduce program on three different datasets and comparing the results.
2. The implementation of K-means clustering on the Wine Dataset (original and custom made extended version) provided by UCI using Spark and Hadoop MapReduce

2. Hadoop and HDFS Setup

I set up Hadoop and HDFS locally on my PC. It was a 1 node setup in Hadoop's pseudo-distributed mode. [Screenshots of Hadoop Web-UI provided below]



Hadoop | Overview | Datanodes | Datanode Volume Failures | Snapshot | Startup Progress | Utilities

Overview 'localhost:9000' (✔active)

Started:	Mon Sep 02 20:08:53 -0400 2024
Version:	3.4.0, rbd8b77f398f626bb7791783192ee7a5dfaeec760
Compiled:	Mon Mar 04 01:35:00 -0500 2024 by root from (HEAD detached at release-3.4.0-RC3)
Cluster ID:	CID-6f22c112-3731-4ef0-9cbc-0ce24d0209ed
Block Pool ID:	BP-682640516-127.0.1.1-1725219113397

Summary

Security is off.

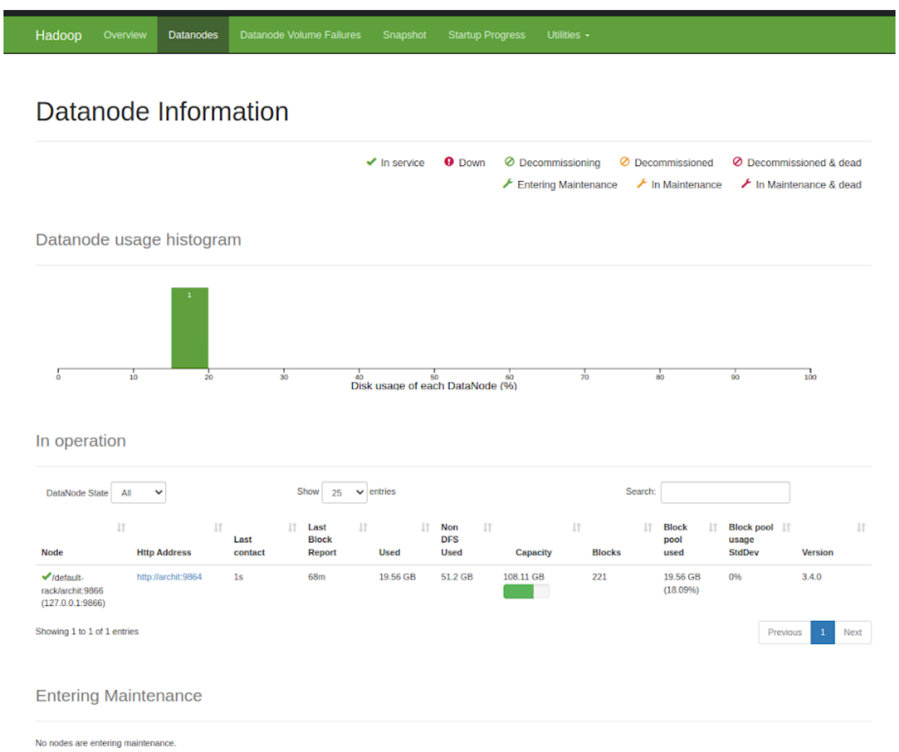
Safemode is off.

106 files and directories, 221 blocks (221 replicated blocks, 0 erasure coded block groups) = 327 total filesystem object(s).

Heap Memory used 26.17 MB of 60.5 MB Heap Memory. Max Heap Memory is 60.5 MB.

Non Heap Memory used 68.68 MB of 70.48 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	108.11 GB
Configured Remote Capacity:	0 B
DFS Used:	19.56 GB (18.09%)
Non DFS Used:	51.2 GB
DFS Remaining:	31.81 GB (29.43%)
Block Pool Used:	19.56 GB (18.09%)
DataNodes usages% (Min/Median/Max/stdDev):	18.09% / 18.09% / 18.09% / 0.00%
Live Nodes	1 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0



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

In operation

DataNode State All | Show 25 entries | Search:

Node	Http Address	Last contact	Last Block Report	Used	Non DFS Used	Capacity	Blocks	Block pool used	Block pool usage StdDev	Version
✔/default-rack/archit:9866 (127.0.0.1:9866)	http://archit:9864	1s	68m	19.56 GB	51.2 GB	108.11 GB	221	19.56 GB (18.09%)	0%	3.4.0

Showing 1 to 1 of 1 entries

Previous 1 Next

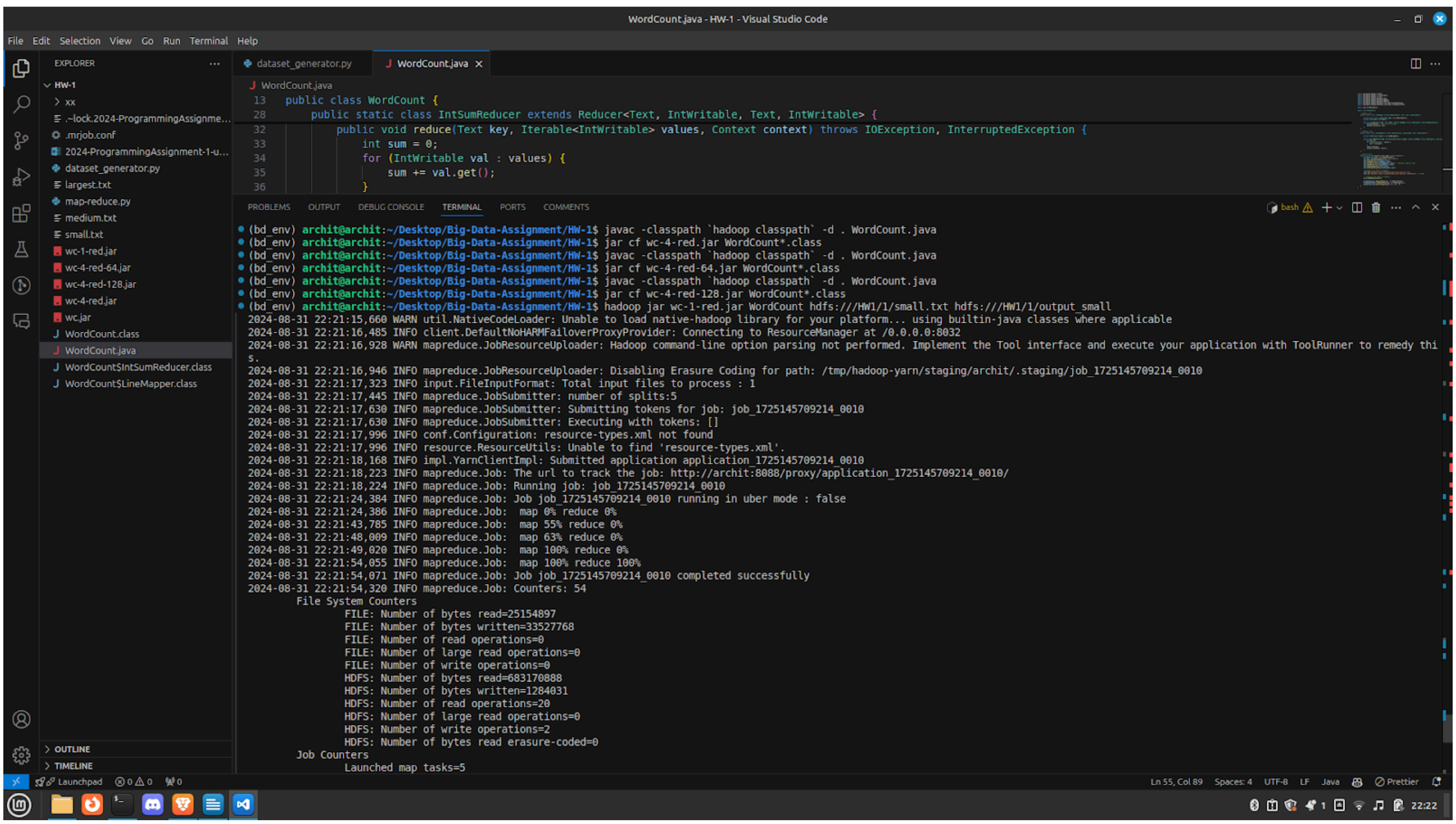
Entering Maintenance

No nodes are entering maintenance.

3. Hadoop MapReduce: Word Count Program

The Word Count MapReduce program was implemented in Hadoop Map Reduce as well as Spark and runtimes for different configs were compared.

Execution Environment: VSCode



Input: Text File with words separated by line breaks

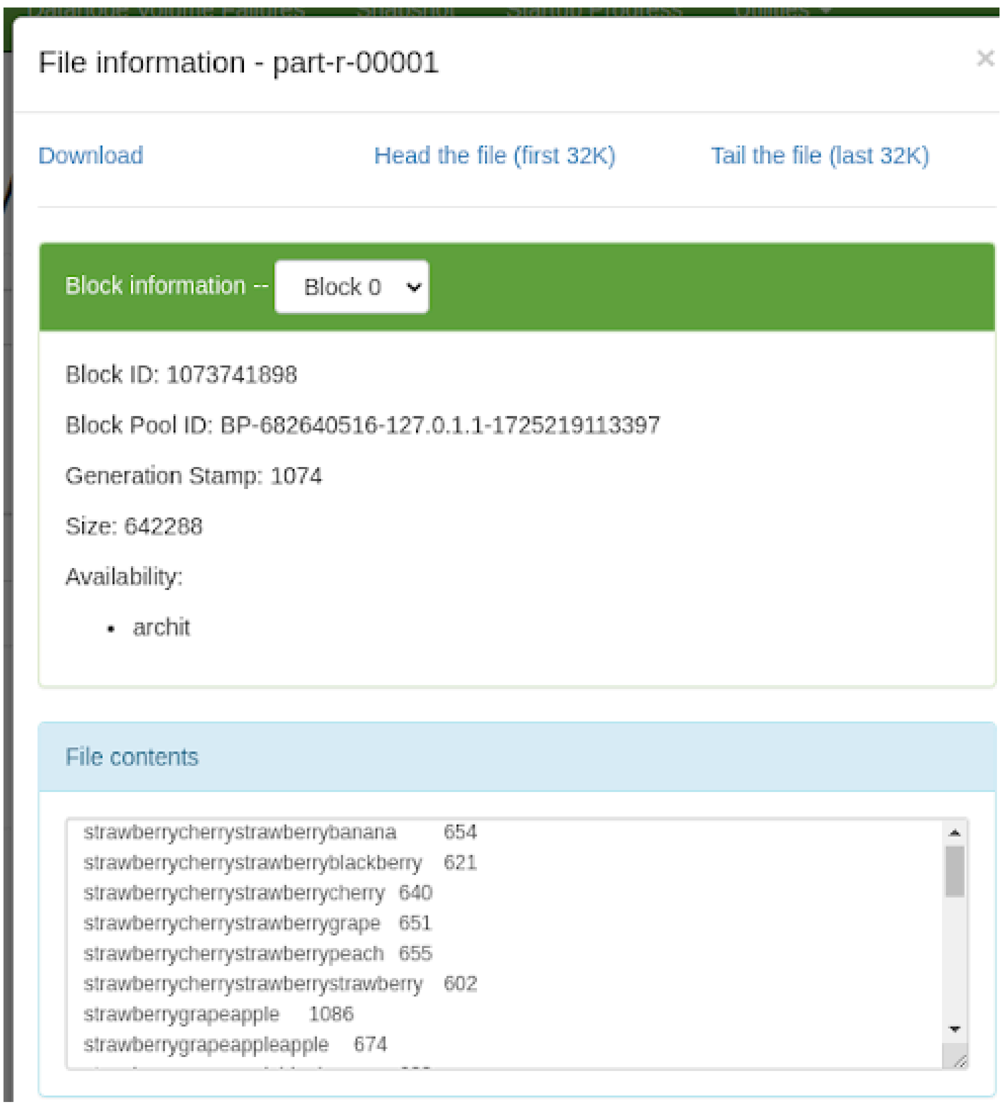
Output: Text File with each word followed by comma separated word count

I ran 4 different scenarios:

Scenario 1: Small Dataset
Size : 700 MB
Input Split Size : 128 MB
Number of Mappers : 6
Number of Reducers : 2

Total time spent by all map tasks (ms)=140366
Total time spent by all reduce tasks (ms)=11171

Output Screenshot:



Scenario 2 : Medium Dataset

Size: 1.68 GB

Input Split Size: 128 MB

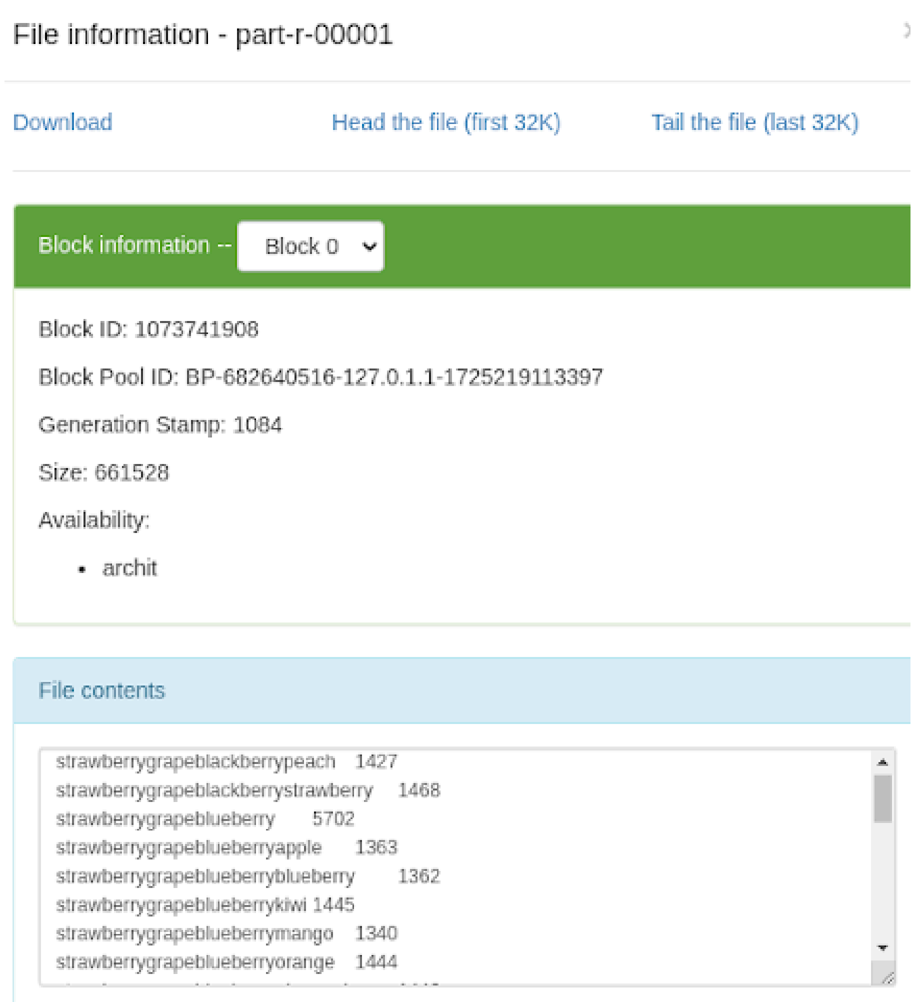
Number of Mappers: 15

Number of Reducers: 2

Total time spent by all map tasks (ms)=301562

Total time spent by all reduce tasks (ms)=56762

Output Screenshot:



Scenario 3a : Large Dataset (2 Reducers)

Size: 9.2 GB

Input Split Size: 128 MB

Number of Mappers: 70

Number of Reducers: 2

Total time spent by all map tasks (ms)=1681110

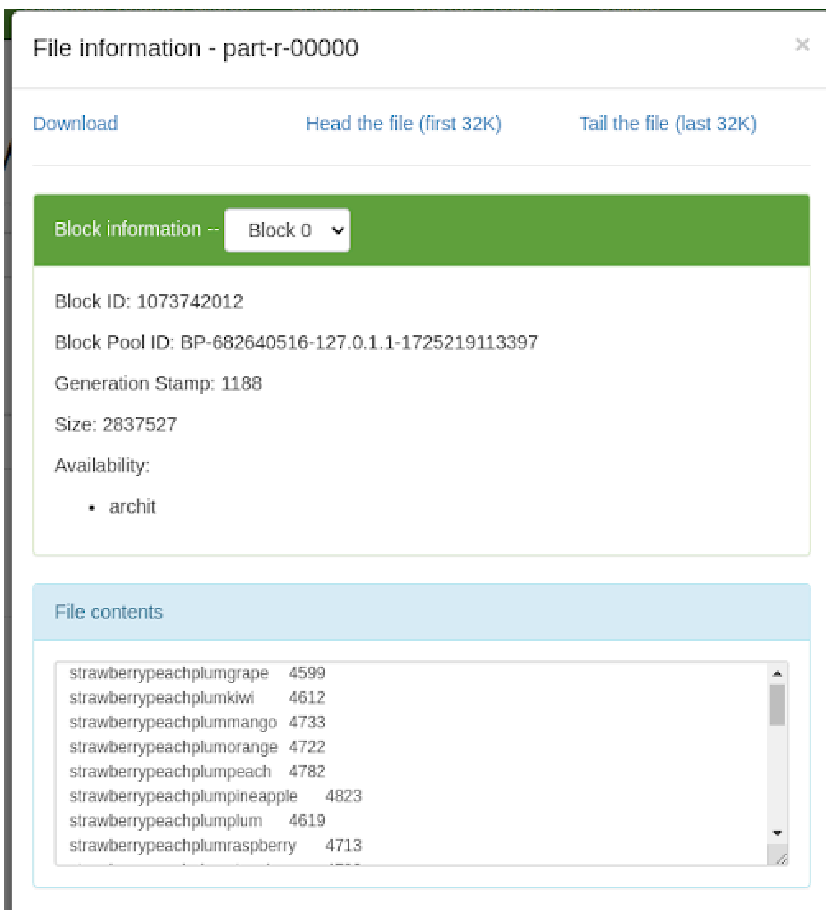
Total time spent by all reduce tasks (ms)=574471

Scenario 3b : Large Dataset (4 Reducers)

Number of Mappers: 69
Number of Reducers: 4

Total time spent by all map tasks (ms)= 2150577

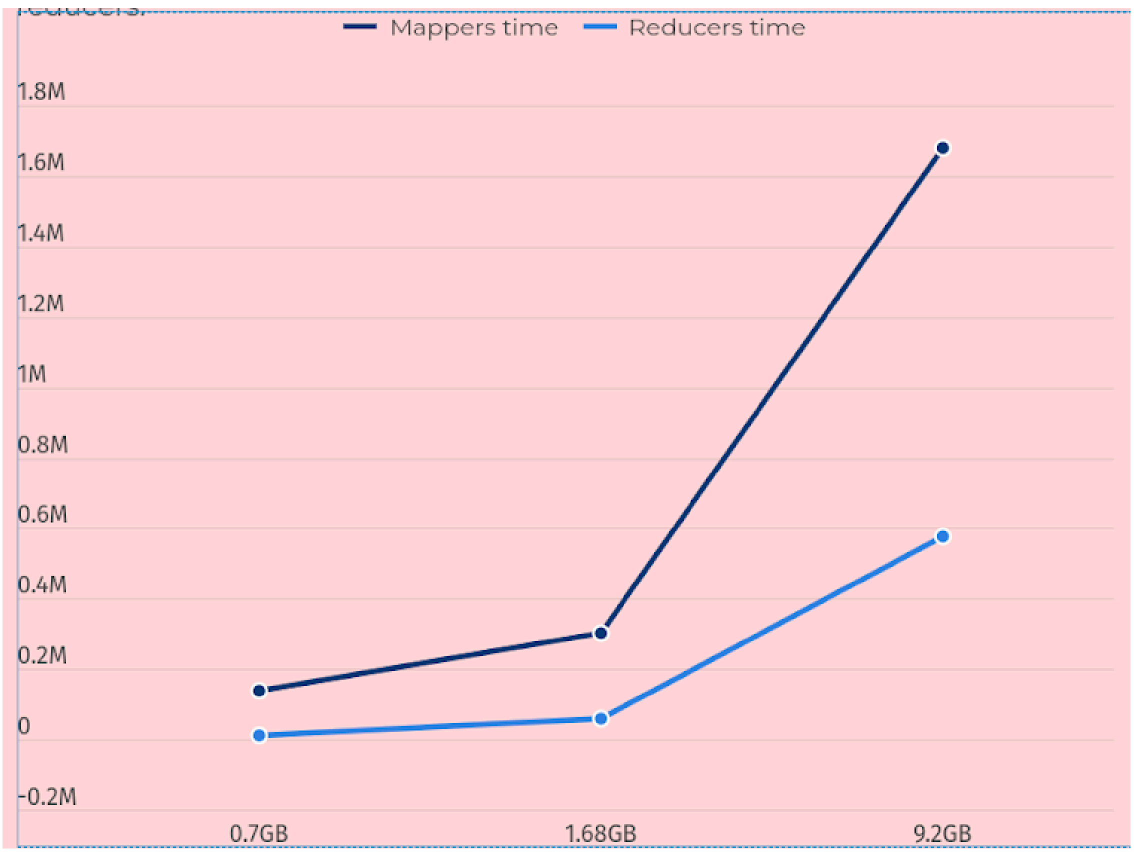
Total time spent by all reduce tasks (ms)= 1120301



Data Comparison 1 : 3 datasets

In this analysis all tasks have been run with 2 reducers and a split size of 128MB

	Small Dataset	Medium Dataset	Large Dataset
Size (GB)	0.7	1.68	9.2
Mappers	6	15	70
Mapper Time (ms)	140366	301562	1681110
Reducer time (ms)	11171	56762	574471



Analysis:

Mapper Time Analysis:

- The mapper time increases significantly with the dataset size.
- Small to Medium Dataset: The mapper time more than doubles (from 140,366 ms to 301,562 ms) when moving from a small to a medium dataset, due to increased data processing load.
- Medium to Large Dataset: The mapper time sees an exponential increase to 16,881,110ms for the large dataset, indicating that the system's resources might be significantly strained at this scale.

Reducer Time Analysis:

- Similar to mapper time, the reducer time also increases as the dataset size grows.
- Small to Medium Dataset: The reducer time increases from 11,171 ms to 56,762 ms, roughly a 5x increase.
- Medium to Large Dataset: The reducer time increases to 574,471 ms for the large dataset, which is a 10x increase from the medium dataset. This disproportionate increase hints at potential bottlenecks in the reduce phase, likely due to limited memory or disk I/O performance.

Mapper to Reducer Ratio:

- Small Dataset: The ratio of mapper time to reducer time is roughly 12.6:1.
- Medium Dataset: The ratio remains somewhat consistent at 5.3:1.
- Large Dataset: The ratio jumps significantly to approximately 29.4:1. Both mapper and reducer tasks are becoming increasingly time consuming. It’s likely that starting up multiple mappers is causing significant overhead.

In a low resource system running multiple mappers and large datasets causes significant overhead , slowing down the entire task. It might be better to stick to a larger split size and lower number of mappers.

Data Comparison 2 : Large Dataset

	2 Reducers	4 Reducers
Mapper Time (ms)	1681110	2150577
Reducer Time (ms)	574471	1120301

Analysis

Mapper Time :

- 2 Reducers: The mapper time is 1,681,110 ms.
- 4 Reducers: The mapper time increases to 2,150,577 ms, suggesting that doubling the number of reducers adds overhead to the mapper phase, possibly due to increased communication or data shuffling.

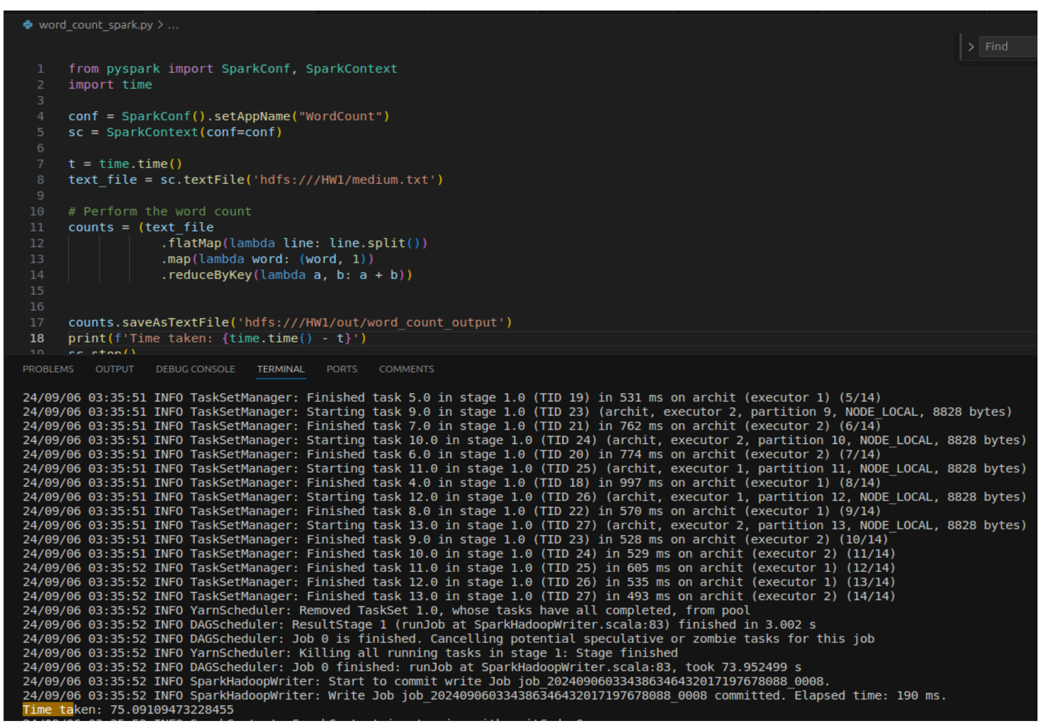
Reducer Time :

- 2 Reducers: Reducer time is 574,471 ms.
- 4 Reducers: Reducer time nearly doubles to 1,120,301 ms. This increase might indicate that the additional reducers are not improving efficiency as expected and might be causing more complex data aggregation processes.

Increasing the number of reducers from 2 to 4 on this large dataset results in longer mapper and reducer times, suggesting that the reducers run one by one and not in parallel. This makes sense because it is a single node Hadoop setup running in pseudo-distributed mode.

Comparing with Spark

The word count program was run in **Spark** with the medium dataset (1.68 GB) . This allows us to compare Spark with Hadoop Map Reduce. For the configuration we used the same mapper/reducer settings as our earlier test.



Time taken by Spark: **75091 ms**
Total time taken by Hadoop = 301562+ 56762 = **358324 ms**

Hadoop Map Reduce	Spark
358324 ms	75091 ms

Analysis

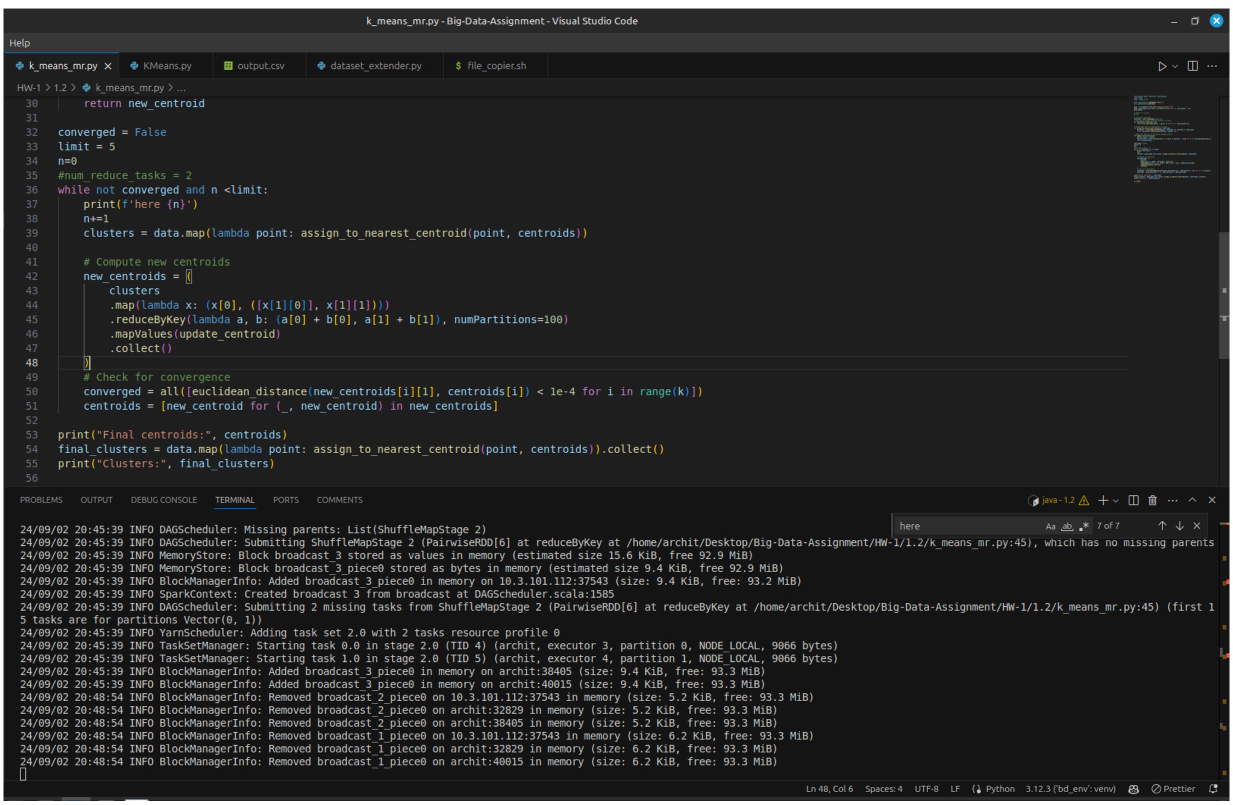
- Spark completed the task in approximately 75.1 seconds, which is significantly faster than Hadoop MapReduce, which took about 358.3 seconds. This demonstrates Spark's superior performance.
- Even though the dataset size is bigger than our heap memory size, Spark's ability to perform in-memory computations can reduce the time needed for data processing tasks.
- This also means that Spark avoids overhead when reading and writing to disk.
- If we had large enough Heap size the time taken by Spark would've been even lesser.

4. K-means Clustering

Implementation:

- I picked the UCI wines database for clustering . It provides detailed chemical analysis of wines derived from different cultivars in the same region in Italy. Source:https://archive.ics.uci.edu/dataset/109/wine
- The dataset has 13 features and 178 instances.
- We scale the dataset down to a normalized range
- Then, since K-Means doesn't work well on higher dimensional data we first use dimensionality reduction to bring it down to 2 dimensions. For this we used Principal Component Analysis .
- Then we run K Means on this reduced dimension dataset using Spark running on Hadoop
- We test different configurations in spark, using different counts for reducers to compare the time taken.
- In order to compare the same program on different sizes of datasets we need to extend this dataset. Since the source only has 178 rows we artificially increase the size of the dataset by adding noise to each of the rows randomly and adding them as new rows. We do this to generate 2 more datasets and run K means on each of them as well.

Execution Environment:



Original Dataset (13 Dimensions)

```
_copier.sh ./    WordCount.java    KMeans.py 5 ●    PCA.py    wine.dat
wine.data

1,14.23,1.71,2.43,15.6,127,2.8,3.06,.28,2.29,5.64,1.04,3.92,1065
1,13.2,1.78,2.14,11.2,100,2.65,2.76,.26,1.28,4.38,1.05,3.4,1050
1,13.16,2.36,2.67,18.6,101,2.8,3.24,.3,2.81,5.68,1.03,3.17,1185
1,14.37,1.95,2.5,16.8,113,3.85,3.49,.24,2.18,7.8,.86,3.45,1480
1,13.24,2.59,2.87,21,118,2.8,2.69,.39,1.82,4.32,1.04,2.93,735
1,14.2,1.76,2.45,15.2,112,3.27,3.39,.34,1.97,6.75,1.05,2.85,1450
1,14.39,1.87,2.45,14.6,96,2.5,2.52,.3,1.98,5.25,1.02,3.58,1290
1,14.06,2.15,2.61,17.6,121,2.6,2.51,.31,1.25,5.05,1.06,3.58,1295
1,14.83,1.64,2.17,14,97,2.8,2.98,.29,1.98,5.2,1.08,2.85,1045
1,13.86,1.35,2.27,16,98,2.98,3.15,.22,1.85,7.22,1.01,3.55,1045
1,14.1,2.16,2.3,18,105,2.95,3.32,.22,2.38,5.75,1.25,3.17,1510
1,14.12,1.48,2.32,16.8,95,2.2,2.43,.26,1.57,5,1.17,2.82,1280
1,13.75,1.73,2.41,16,89,2.6,2.76,.29,1.81,5.6,1.15,2.9,1320
1,14.75,1.73,2.39,11.4,91,3.1,3.69,.43,2.81,5.4,1.25,2.73,1150
1,14.38,1.87,2.38,12,102,3.3,3.64,.29,2.96,7.5,1.2,3,1547
1,13.63,1.81,2.7,17.2,112,2.85,2.91,.3,1.46,7.3,1.28,2.88,1310
1,14.3,1.92,2.72,20,120,2.8,3.14,.33,1.97,6.2,1.07,2.65,1280
1,13.83,1.57,2.62,20,115,2.95,3.4,.4,1.72,6.6,1.13,2.57,1130
1,14.19,1.59,2.48,16.5,108,3.3,3.93,.32,1.86,8.7,1.23,2.82,1680
1,13.64,3.1,2.56,15.2,116,2.7,3.03,.17,1.66,5.1,.96,3.36,845
1,14.06,1.63,2.28,16,126,3,3.17,.24,2.1,5.65,1.09,3.71,780
1,12.93,3.8,2.65,18.6,102,2.41,2.41,.25,1.98,4.5,1.03,3.52,770
1,13.71,1.86,2.36,16.6,101,2.61,2.88,.27,1.69,3.8,1.11,4,1035
1,12.85,1.6,2.52,17.8,95,2.48,2.37,.26,1.46,3.93,1.09,3.63,1015
1,13.5,1.81,2.61,20,96,2.53,2.61,.28,1.66,3.52,1.12,3.82,845
1,13.05,2.05,3.22,25,124,2.63,2.68,.47,1.92,3.58,1.13,3.2,830
1,13.39,1.77,2.62,16.1,93,2.85,2.94,.34,1.45,4.8,.92,3.22,1195
1,13.3,1.72,2.14,17,94,2.4,2.19,.27,1.35,3.95,1.02,2.77,1285
1,13.87,1.9,2.8,19.4,107,2.95,2.97,.37,1.76,4.5,1.25,3.4,915
1,14.02,1.68,2.21,16,96,2.65,2.33,.26,1.98,4.7,1.04,3.59,1035
1,13.73,1.5,2.7,22.5,101,3,3.25,.29,2.38,5.7,1.19,2.71,1285
1,13.58,1.66,2.36,19.1,106,2.86,3.19,.22,1.95,6.9,1.09,2.88,1515
1,13.68,1.83,2.36,17.2,104,2.42,2.69,.42,1.97,3.84,1.23,2.87,990
1,13.76,1.53,2.7,19.5,132,2.95,2.74,.5,1.35,5.4,1.25,3,1235
1,13.51,1.8,2.65,19,110,2.35,2.53,.29,1.54,4.2,1.1,2.87,1095
1,13.48,1.81,2.41,20.5,100,2.7,2.98,.26,1.86,5.1,1.04,3.47,920
1,13.28,1.64,2.84,15.5,110,2.6,2.68,.34,1.36,4.6,1.09,2.78,880
1,13.05,1.65,2.55,18,98,2.45,2.43,.29,1.44,4.25,1.12,2.51,1105
```

Dataset after PCA

```
             0          1
0   -10.105899 -15.083798
1    -9.001728 -13.311941
2    -9.308139 -14.672647
3   -10.544975 -16.393015
4    -7.804549 -14.511780
..         ...        ...
173  -3.437435 -15.854451
174  -4.203841 -15.396683
175  -4.128171 -16.397529
176  -4.418176 -15.935281
177  -3.598746 -16.405527

[178 rows x 2 columns]
```

Input: CSV file with features

Output: Cluster centers as tuples, and the clusters as a list of arrays, 1 for each datapoint and its cluster center index.

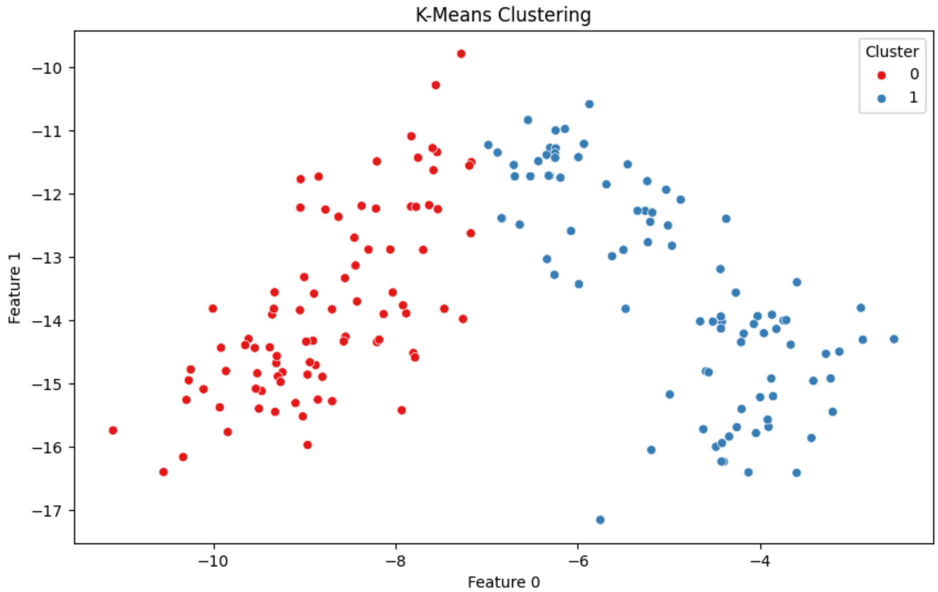
A) Initial Configuration

*The following runs were using **2 cores, 2 executors and 3 reducers** on spark*

k=2

Time Taken: 30.65s

Clustering plot



Output

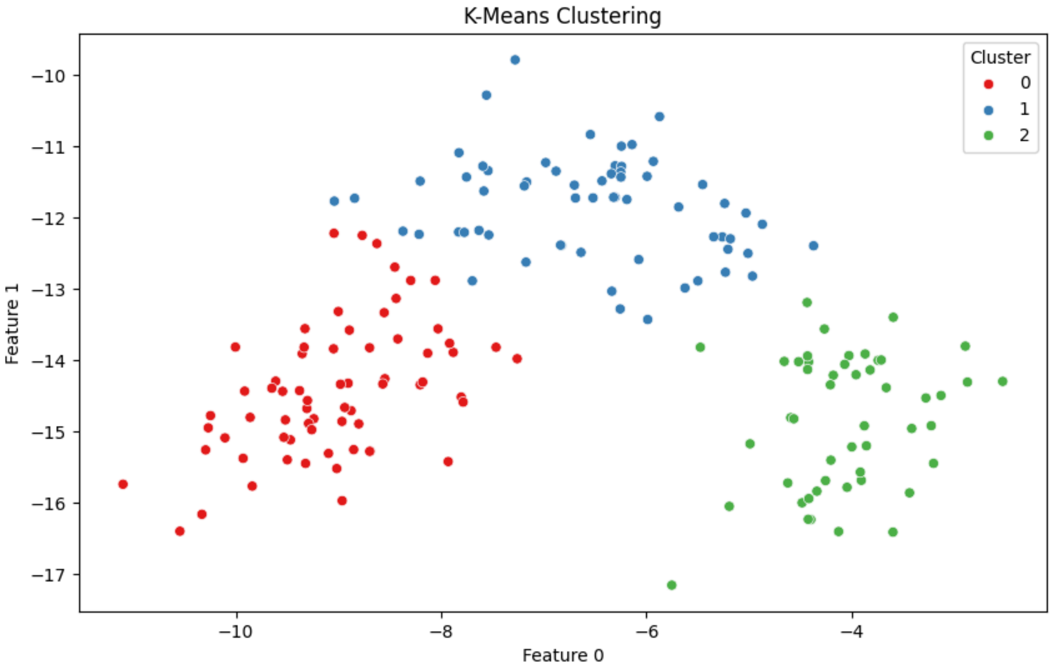
```
Final centroids:
 [(-8.665072384606301, -13.701838037384848), (-4.846063475536938, -13.584313304884258)]
```

```
Clusters:
 [[0, ([array([-10.10589912, -15.08379843])], 1)], [0, ([array([ -9.00172796, -13.3119411 ])], 1)], [0, ([array([ -9.30813884, -14.6726469
 [0, ([array([ -7.80454863, -14.51178021])], 1)], [0, ([array([ -9.84015187, -15.76082709])], 1)], [0, ([array([ -9.24067866, -14.81594152]
[0, ([array([ -9.30228949, -14.55988465])], 1)], [0, ([array([ -9.54436053, -14.4316132 ])], 1)], [0, ([array([-10.26842652, -14.94306602])
0, ([array([ -8.90599543, -14.31820179])], 1)], [0, ([array([-10.24690736, -14.77184565])], 1)], [0, ([array([-11.09913042, -15.73447588])]
, ([array([ -8.96432383, -15.96515493])], 1)], [0, ([array([ -8.69210762, -15.2711761 ])], 1)], [0, ([array([-10.33049981, -16.15565588])],
 ([array([ -9.9140919 , -14.42907978])], 1)], [0, ([array([ -7.88199175, -13.88545971])], 1)], [0, ([array([ -9.32657078, -13.5528139 ])],
([array([ -8.5550988 , -13.32813936])], 1)], [0, ([array([ -7.78577221, -14.58241747])], 1)], [0, ([array([ -8.56876201, -14.32864114])], 1
[array([ -8.98072861, -14.33202607])], 1)], [0, ([array([ -9.04824081, -13.83531954])], 1)], [0, ([array([ -9.29166519, -14.88173961])], 1)
array([ -8.42247616, -13.69695237])], 1)], [0, ([array([ -8.69581684, -15.2728629 ])], 1)], [0, ([array([ -8.20489932, -14.34036725])], 1)]
rray([ -8.17944483, -14.30118331])], 1)], [0, ([array([ -7.91752885, -13.75818495])], 1)], [0, ([array([ -8.29644705, -12.87712854])], 1)],
ray([ -9.3792934 , -14.42181946])], 1)], [0, ([array([ -7.46507973, -13.81387902])], 1)], [0, ([array([ -9.86064712, -14.79706359])], 1)],
ay([ -8.89391909, -13.573587  ])], 1)], [0, ([array([ -7.93144835, -15.41650466])], 1)], [0, ([array([ -9.51740933, -14.83237971])], 1)], [
y([ -8.80267539, -14.88791781])], 1)], [0, ([array([ -9.49835342, -15.39143539])], 1)], [0, ([array([-10.00435223, -13.81091843])], 1)], [0
([-10.29422143, -15.2525935 ])], 1)], [0, ([array([ -9.01701131, -15.51428943])], 1)], [0, ([array([ -8.93942661, -14.65828765])], 1)], [0,
[ -9.53228432, -15.07690402])], 1)], [0, ([array([ -8.96610445, -14.85318616])], 1)], [0, ([array([ -9.92902739, -15.37110445])], 1)], [1,
 -5.26033692, -12.26683859])], 1)], [1, ([array([ -4.96739365, -12.81674675])], 1)], [0, ([array([ -6.82899888, -12.38516218])], 1)], [0, (
-6.19051231, -11.74170463])], 1)], [0, ([array([ -7.69617369, -12.88263354])], 1)], [0, ([array([ -9.04066041, -11.76510494])], 1)], [0, ([
5.98795949, -13.42450248])], 1)], [0, ([array([ -8.76854133, -12.24506031])], 1)], [1, ([array([ -5.23068448, -12.76190245])], 1)], [0, ([a
.07514619, -12.58374275])], 1)], [0, ([array([ -9.35349791, -13.9038628 ])], 1)], [0, ([array([ -8.6258908 , -12.36014071])], 1)], [1, ([ar
16745172, -11.49654802])], 1)], [1, ([array([ -5.34520062, -12.2649357 ])], 1)], [0, ([array([ -8.05785644, -12.875715  ])], 1)], [0, ([arr
583984, -10.2788362])], 1)], [0, ([array([ -7.8301459, -12.1977672])], 1)], [1, ([array([ -6.30499345, -11.26985095])], 1)], [1, ([array([
58, -12.17487098])], 1)], [0, ([array([ -7.58416643, -11.62347044])], 1)], [1, ([array([ -5.99391556, -11.41684943])], 1)], [1, ([array([ -
5, -11.8472186 ])], 1)], [1, ([array([ -6.24431853, -10.99433165])], 1)], [1, ([array([ -5.45298833, -11.53235253])], 1)], [1, ([array([ -5
, -12.08928749])], 1)], [0, ([array([ -7.54304375, -11.33797061])], 1)], [0, ([array([ -7.75324024, -11.4271224 ])], 1)], [0, ([array([ -9.
 -13.2765074 ])], 1)], [0, ([array([ -7.82662761, -11.08604363])], 1)], [0, ([array([ -9.04405307, -12.21568502])], 1)], [0, ([array([ -8.2
-11.2741336 ])], 1)], [1, ([array([ -6.25049222, -11.35771798])], 1)], [1, ([array([ -6.63775778, -12.48318848])], 1)], [1, ([array([ -6.14
11.55156808])], 1)], [1, ([array([ -5.03096501, -11.93193834])], 1)], [1, ([array([ -6.43324108, -11.48114515])], 1)], [1, ([array([ -5.182
```

k=3

Time Taken : 34.92s

Clustering plot



Output

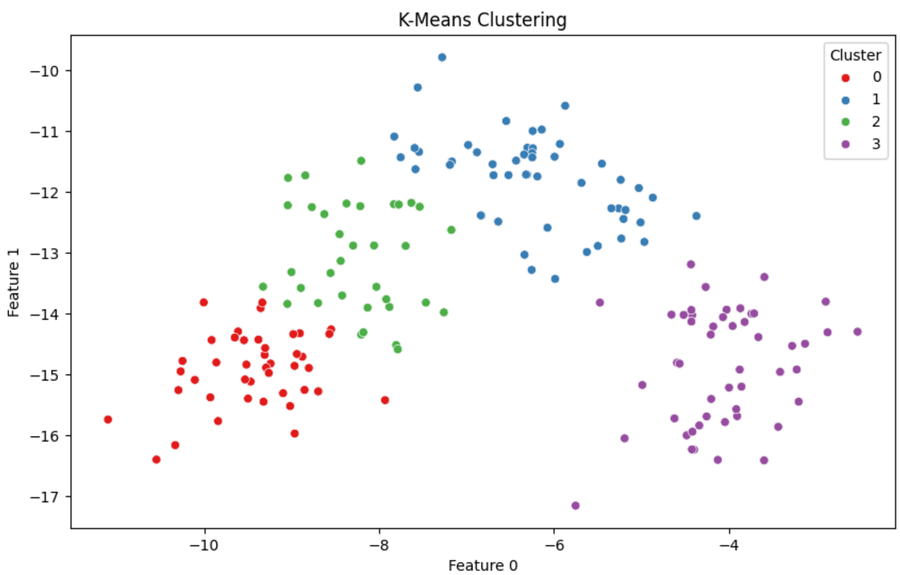
```
Final centroids:
 [(-4.068467701955728, -14.805883724834294), (-9.080921748391471, -14.601643135772887), (-6.757753695839114, -11.90609106571778)]
```

```
Clusters:
 [[1, ([array([-10.10589912, -15.08379843])], 1)], [1, ([array([ -9.00172796, -13.3119411 ])], 1)], [1, ([array([ -9.30813884, -14.6726469 ])], 1)], [1, ([array([-10.54497
 [1, ([array([ -7.80454863, -14.51178021])], 1)], [1, ([array([ -9.84015187, -15.76082709])], 1)], [1, ([array([ -9.24067866, -14.81594152])], 1)], [1, ([array([ -8.852121
[1, ([array([ -9.30228949, -14.55988465])], 1)], [1, ([array([ -9.54436053, -14.4316132 ])], 1)], [1, ([array([-10.26842652, -14.94306602])], 1)], [1, ([array([ -8.5482950
1, ([array([ -8.90599543, -14.31820179])], 1)], [1, ([array([-10.24690736, -14.77184565])], 1)], [1, ([array([-11.09913042, -15.73447588])], 1)], [1, ([array([ -9.09718198
, ([array([ -8.96432383, -15.96515493])], 1)], [1, ([array([ -8.69210762, -15.2711761 ])], 1)], [1, ([array([-10.33049981, -16.15565588])], 1)], [1, ([array([ -8.8771367 ,
 ([array([ -9.9140919 , -14.42907978])], 1)], [1, ([array([ -7.88199175, -13.88545971])], 1)], [1, ([array([ -9.32657078, -13.5528139 ])], 1)], [1, ([array([ -8.43883922,
([array([ -8.5550988 , -13.32813936])], 1)], [1, ([array([ -7.78577221, -14.58241747])], 1)], [1, ([array([ -8.56876201, -14.32864114])], 1)], [1, ([array([ -8.03042693, -
[array([ -8.98072861, -14.33202607])], 1)], [1, ([array([ -9.04824081, -13.83531954])], 1)], [1, ([array([ -9.29166519, -14.88173961])], 1)], [1, ([array([ -9.46835778, -1
array([ -8.42247616, -13.69695237])], 1)], [1, ([array([ -8.69581684, -15.2728629 ])], 1)], [1, ([array([ -8.20489932, -14.34036725])], 1)], [1, ([array([ -8.69694902, -13
rray([ -8.17944483, -14.30118331])], 1)], [1, ([array([ -7.91752885, -13.75818495])], 1)], [2, ([array([ -8.29644705, -12.87712854])], 1)], [1, ([array([ -9.32116304, -15.
ray([ -9.3792934 , -14.42181946])], 1)], [1, ([array([ -7.46507973, -13.81387902])], 1)], [1, ([array([ -9.86064712, -14.79706359])], 1)], [1, ([array([ -7.25938711, -13.9
ay([ -8.89391909, -13.573587  ])], 1)], [1, ([array([ -7.93144835, -15.41650466])], 1)], [1, ([array([ -9.51740933, -14.83237971])], 1)], [1, ([array([ -9.61188114, -14.28
y([ -8.80267539, -14.88791781])], 1)], [1, ([array([ -9.49835342, -15.39143539])], 1)], [1, ([array([-10.00435223, -13.81091843])], 1)], [1, ([array([ -9.64939587, -14.387
([-10.29422143, -15.2525935 ])], 1)], [1, ([array([ -9.01701131, -15.51428943])], 1)], [1, ([array([ -8.93942661, -14.65828765])], 1)], [1, ([array([ -9.26086154, -14.9696
[ -9.53228432, -15.07690402])], 1)], [1, ([array([ -8.96610445, -14.85318616])], 1)], [1, ([array([ -9.92902739, -15.37110445])], 1)], [2, ([array([ -5.87250822, -10.57955
 -5.26033692, -12.26683859])], 1)], [2, ([array([ -4.96739365, -12.81674675])], 1)], [2, ([array([ -6.82899888, -12.38516218])], 1)], [2, ([array([ -8.84297248, -11.724778
-6.19051231, -11.74170463])], 1)], [2, ([array([ -7.69617369, -12.88263354])], 1)], [2, ([array([ -9.04066041, -11.76510494])], 1)], [2, ([array([ -6.98134632, -11.2240774
```

k=4

Time Taken: 36.33s

Clustering plot:



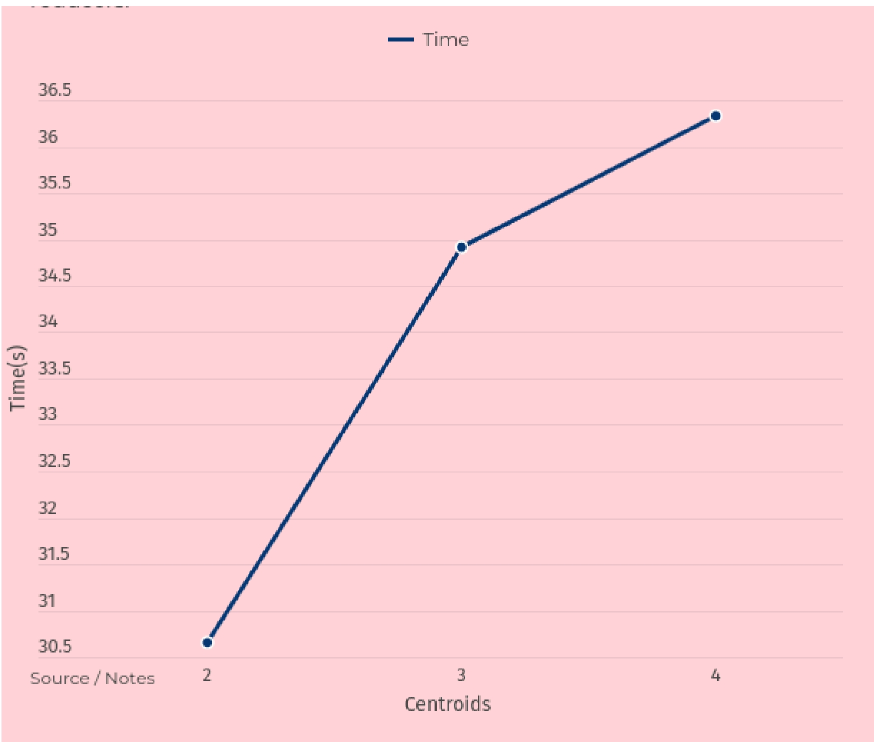
Output:

```
Final centroids:
 [(-9.080921748391468, -14.601643135772889), (-5.359318693352522, -12.576554873739507), (-7.234826511206128, -11.693706139975008), (-4.024270023606468, -14.912830519946818)]
```

```
Clusters:
 [[0, ([array([-10.10589912, -15.08379843])], 1)], [0, ([array([ -9.00172796, -13.3119411 ])], 1)], [0, ([array([ -9.30813884, -14.
 [0, ([array([ -7.80454863, -14.51178021])], 1)], [0, ([array([ -9.84015187, -15.76082709])], 1)], [0, ([array([ -9.24067866, -14.8
[0, ([array([ -9.30228949, -14.55988465])], 1)], [0, ([array([ -9.54436053, -14.4316132 ])], 1)], [0, ([array([-10.26842652, -14.94
0, ([array([ -8.90599543, -14.31820179])], 1)], [0, ([array([-10.24690736, -14.77184565])], 1)], [0, ([array([-11.09913042, -15.734
, ([array([ -8.96432383, -15.96515493])], 1)], [0, ([array([ -8.69210762, -15.2711761 ])], 1)], [0, ([array([-10.33049981, -16.1556
 ([array([ -9.9140919 , -14.42907978])], 1)], [0, ([array([ -7.88199175, -13.88545971])], 1)], [0, ([array([ -9.32657078, -13.55281
([array([ -8.5550988 , -13.32813936])], 1)], [0, ([array([ -7.78577221, -14.58241747])], 1)], [0, ([array([ -8.56876201, -14.328641
[array([ -8.98072861, -14.33202607])], 1)], [0, ([array([ -9.04824081, -13.83531954])], 1)], [0, ([array([ -9.29166519, -14.8817396
array([ -8.42247616, -13.69695237])], 1)], [0, ([array([ -8.69581684, -15.2728629 ])], 1)], [0, ([array([ -8.20489932, -14.34036725
rray([ -8.17944483, -14.30118331])], 1)], [0, ([array([ -7.91752885, -13.75818495])], 1)], [2, ([array([ -8.29644705, -12.87712854]
ray([ -9.3792934 , -14.42181946])], 1)], [0, ([array([ -7.46507973, -13.81387902])], 1)], [0, ([array([ -9.86064712, -14.79706359])
ay([ -8.89391909, -13.573587  ])], 1)], [0, ([array([ -7.93144835, -15.41650466])], 1)], [0, ([array([ -9.51740933, -14.83237971])]
y([ -8.80267539, -14.88791781])], 1)], [0, ([array([ -9.49835342, -15.39143539])], 1)], [0, ([array([-10.00435223, -13.81091843])],
([-10.29422143, -15.2525935 ])], 1)], [0, ([array([ -9.01701131, -15.51428943])], 1)], [0, ([array([ -8.93942661, -14.65828765])],
[ -9.53228432, -15.07690402])], 1)], [0, ([array([ -8.96610445, -14.85318616])], 1)], [0, ([array([ -9.92902739, -15.37110445])], 1
 -5.26033692, -12.26683859])], 1)], [1, ([array([ -4.96739365, -12.81674675])], 1)], [2, ([array([ -6.82899888, -12.38516218])], 1)
-6.19051231, -11.74170463])], 1)], [2, ([array([ -7.69617369, -12.88263354])], 1)], [2, ([array([ -9.04066041, -11.76510494])], 1)]
5.98795949, -13.42450248])], 1)], [2, ([array([ -8.76854133, -12.24506031])], 1)], [1, ([array([ -5.23068448, -12.76190245])], 1)],
.07514619, -12.58374275])], 1)], [0, ([array([ -9.35349791, -13.9038628 ])], 1)], [2, ([array([ -8.6258908 , -12.36014071])], 1)],
16745172, -11.49654802])], 1)], [1, ([array([ -5.34520062, -12.2649357 ])], 1)], [2, ([array([ -8.05785644, -12.875715  ])], 1)], [
583984, -10.2788362])], 1)], [2, ([array([ -7.8301459, -12.1977672])], 1)], [2, ([array([ -6.30499345, -11.26985095])], 1)], [3, ([
58, -12.17487098])], 1)], [2, ([array([ -7.58416643, -11.62347044])], 1)], [2, ([array([ -5.99391556, -11.41684943])], 1)], [2, ([a
5, -11.8472186 ])], 1)], [2, ([array([ -6.24431853, -10.99433165])], 1)], [1, ([array([ -5.45298833, -11.53235253])], 1)], [1, ([ar
, -12.08928749])], 1)], [2, ([array([ -7.54304375, -11.33797061])], 1)], [2, ([array([ -7.75324024, -11.4271224 ])], 1)], [0, ([arr
 -13.2765074 ])], 1)], [2, ([array([ -7.82662761, -11.08604363])], 1)], [2, ([array([ -9.04405307, -12.21568502])], 1)], [2, ([arra
-11.2741336 ])], 1)], [2, ([array([ -6.25049222, -11.35771798])], 1)], [2, ([array([ -6.63775778, -12.48318848])], 1)], [2, ([array
```

Runtime Comparisons for different number of centroids:

k	time(s)
2	30.65
3	34.92
4	36.33



Analysis:

As expected, the time taken to perform K-means clustering increases with the number of centroids. This behavior aligns with the theoretical understanding of the algorithm, as the computational complexity of K-means is directly proportional to the number of centroids. Specifically, for each iteration of the algorithm, the distance between each data point and all k centroids must be calculated. Therefore, as k increases, the number of distance calculations increases, leading to a corresponding increase in computational time.

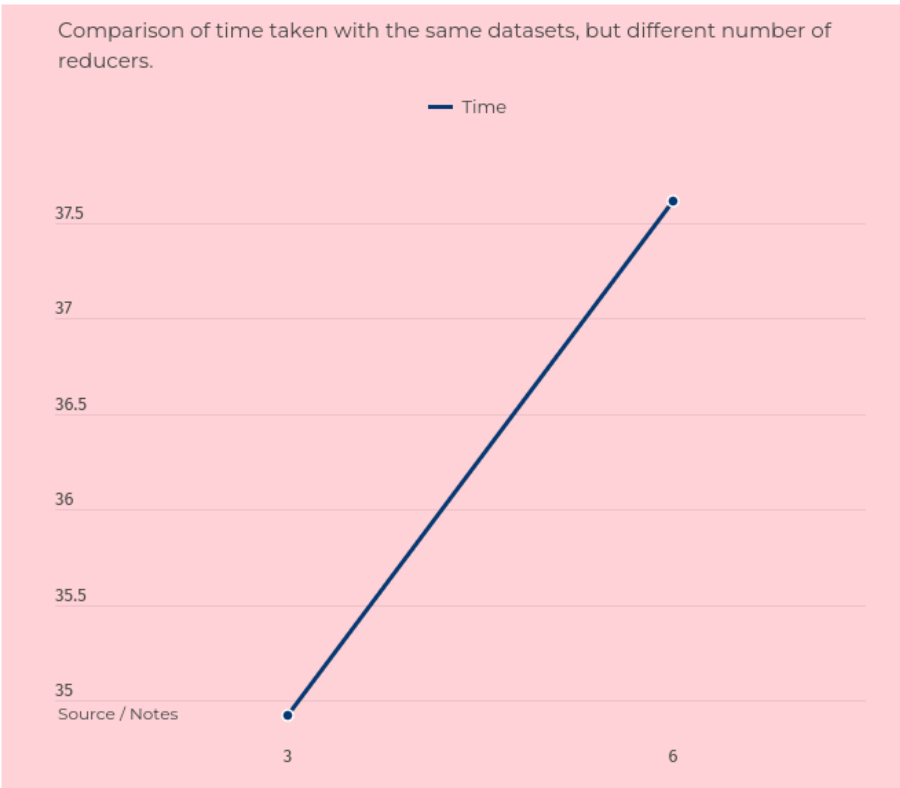
Additionally, the increase in time can also be attributed to the greater number of centroid updates required with more centroids. Each iteration involves recalculating the positions of all k centroids based on the assigned data points, which becomes more computationally intensive as k grows.

B) Alternative Configurations

1) *A run was made by setting **k=3** and changing the **number of reducers to 6** , all other configs stay the same.*

time taken 37.61

Reducers	time(s)
3	34.92
6	37.61

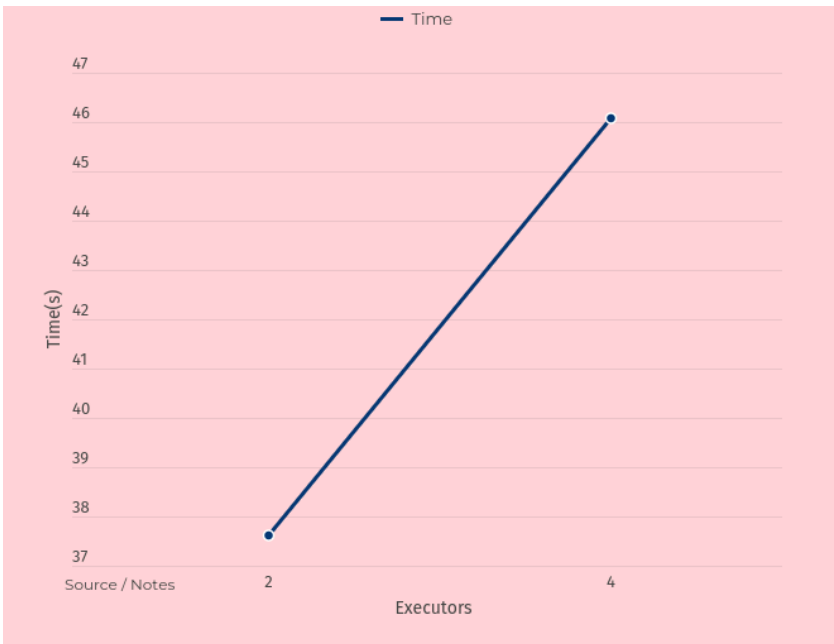


Analysis:
Earlier in the word count example we saw that increasing the number of reducers actually increases the amount of time taken in a single node, low resource Hadoop setup. Here we observe similar results after integrating Spark. However the increase in time is not as significant as it was earlier with time taken increasing by only ~2s . This is probably because Spark handles task parallelism and resource management more efficiently than Hadoop MapReduce, especially when running in a low-resource environment.

2) *A run was made by changing the* ***number of executors used to 4 (k=3) (Reducers = 6)***

Time Taken: 46.08

Executors	t(s)
2	37.61
4	46.08



Analysis:
If we increase the number of executors the time taken increases by a large amount. Each executor requires additional CPU overhead to startup and process tasks. This overhead is negligible on high resource , high volume systems, but on a low resource system it is significant. Thus we can see the time taken to process the dataset increase.

C) Alternative Datasets

The same script was run using 2 more extended datasets, with increased size . Both were run using **3 reducers, 2 cores and 2 executors** .

1) ***Dataset 1*** *: 1.8 MB ~ 45,000 instances*

   Time Taken: 48.03

   Analysis

   - As expected , the time taken does increase compared to ~170 instances. But, the th increase is not proportional to the increase in instances.
     I.e. even though the number of instances increases 264x , the time taken merely increases by 1.45x .
   - This displays Spark's excellent parallelism. Even with a larger dataset, Spark can parallelize the workload across the available executors, thus minimizing the additional time required to process the extra data.
   - This also shows a majority of the time is actually spent in overhead which includes starting up the executors and reducers and such. The actual computation for small seized datasets like these happens relatively quickly.
   - For the smaller dataset, the job might have been more **IO-bound**, where the overhead related to data input/output was significant. For the larger dataset, since more of the data is processed in parallel across multiple cores and executors, the computation could become more **CPU-bound**, utilizing the processing power more effectively without a linear increase in IO operations.

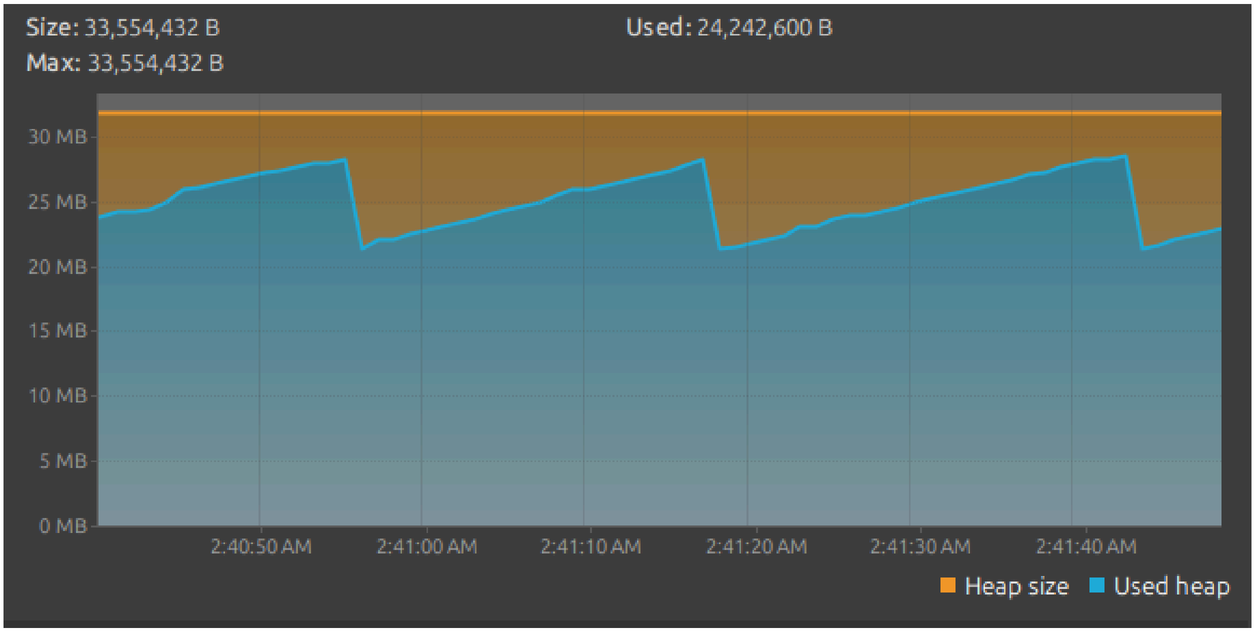
2) ***Dataset 2****: 40 MB :>1 Million Instances*

   Time Taken: 487.34

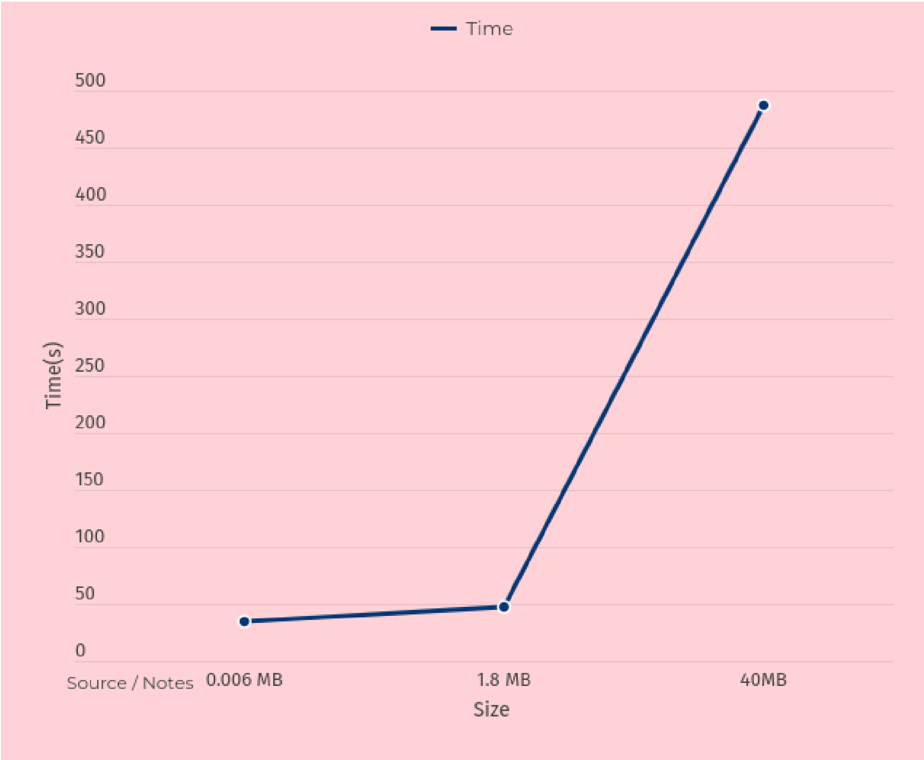
   Analysis:

   In this scenario, the dataset size (40 MB) exceeds the Hadoop heap size (JVM queue size) limit of 32 MB. This is a critical factor that significantly impacts the time taken for processing

   - Since the dataset size exceeds the 32 MB heap size, Hadoop will no longer be able to store all the data in memory. When the heap limit is reached, Hadoop starts spilling data to disk.
   - Intermediate data (e.g., shuffle outputs, map outputs) that cannot fit in memory is written to disk, read back, and written again for further processing.
   - Disk I/O operations are significantly slower than in-memory processing, causing a considerable slowdown in execution time
   - As data spills to disk, there is a higher reliance on disk I/O operations, which are considerably slower than memory operations



As we can see in the above image, as soon as the heap size is close to being full the data is spills over to disk, illustrating the issue with the large dataset.



D) Alternative Solution:

An alternative way to solve K Means is by **broadcasting the centroids** to all workers(mappers) after every iteration. This will reduce overhead by avoiding large data shuffles.

Dataset 1: *0.006 MB ~ 170 instances*
Time Taken: 10.3s

Dataset 2 *: 1.8 MB ~ 45,000 instances*
Time Taken: 19.84s

	Naive Map-Reduce	**Broadcast**
0.006 MB	34.92s	28.03
1.8 MB	48.03s	32.22

Analysis:

- In the naive approach, centroids are passed as part of each task to the worker nodes. Since the data is processed in parallel across multiple nodes, each node needs to receive its own copy of the centroids. This leads to a lot of overhead.
- In the optimized broadcast approach, the centroids are broadcast to all worker nodes once, and the workers store them locally. This drastically reduces the need for repeated network communication during the iterative process and is thus a lot faster. This is especially apparent for the larger dataset.

5. Conclusion

This assignment demonstrated the implementation of Hadoop Map Reduce and Spark for 2 problems - Word Count and K-means clustering The experiments provided valuable insights into performance optimization in distributed computing environments.

Key Takeaways:

- Spark in almost every case will perform better than Hadoop Map Reduce simply because of in-memory computations.
- For single-node, low resource systems having a lesser number of mappers and reducers might be beneficial because it reduces the overhead. Fewer mappers and reducers may lead to lower task-switching and disk I/O costs
- Spark scales better than Hadoop MapReduce for large datasets. With its distributed in-memory processing. However, it still deals with bottlenecks due to JVM queue sizes.

6. References

1) https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html
2) https://archive.ics.uci.edu/dataset/109/wine