CSE 587- Data Intensive Computing
Project II

Big Data Content Retrieval , Storage and Analysis

Anand Nandan Menon
UB # : 50098027
Email: anandnan@buffalo.edu

Subhranil Banerjee
UB # : 50096800
Email: subhrani@buffalo.edu

Abstract:

Big data is the term for a collection of data sets so large and complex that it becomes difficult to process using on-hand database management tools or traditional data processing applications. The challenges include capture, curation, storage, search, sharing, transfer, analysis and visualization. The trend to larger data sets is due to the additional information derivable from analysis of a single large set of related data, as compared to separate smaller sets with the same total amount of data, allowing correlations to be found to "spot business trends, determine quality of research, prevent diseases, link legal citations, combat crime, and determine real-time roadway traffic conditions."

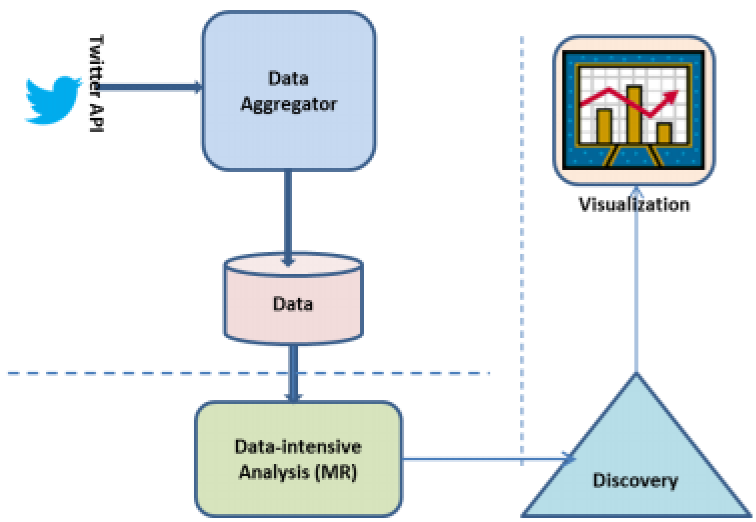
In this project, we shift our focus to one of the most popular websites of the world, Twitter, a gigantic hub of data accessed and modified , every second by millions of users all over the world. To analyze this data, we utilize the power of parallel processing given to us by the Mapreduce programming model.

Objective:

To analyze Twitter data using the Mapreduce programming model and using various visualization tools on the output .

Project Approach:

Our approach can be summarized in three major steps. The following figure further enunciates them.



- Collecting data from twitter's live feed using the Streaming API's provided.
- Feeding this data to a Hadoop 2.20 cluster for analysis.
- Visualizing the output using various tools like R and Gephi.

Data Aggregation:

For the purpose of this project , we used the *TwitterStream* Api's provided by twitter to get data off the public feed.

Aggregator: Our custom application *TweetAggregator* uses the above mentioned api's by way of a public and secret key pair . It implements a listener class for receiving notifications whenever a tweet is captured. For every tweet, the following information is maintained.

- Text of the tweet.
- Screen name of user who tweeted.
- Follower Count of the user.

This data is stored locally on files. Each such file contain 10000 tweets. The number 10000 was chosen as a tradeoff between the size of an individual file (to avoid exceeding the JVM's heap for in-mapper combining) and the number of mappers.

Cleaning: Our custom application *TweetCleaner* takes each of the above mentioned files as input and for every line (tweet) therein, removes punctuations , stop words and extra spaces . The output of this application is used as input for our mapreduce jobs.

Volume of Data Collected: We collected a total of 1.75 million tweets in two sets of 1 million and 750k respectively. However the results from the first set were discarded as they were deemed "uninteresting".

<u>Goals:</u>

1.WordCount:

<u>Requirement Specification:</u> To find out the most trending words, hashtags and mentions.
<u>Implementation Details:</u> The algorithm comprises two parts:

- The first phase implements a similar logic for wordcount as given in chapter 4 of Lin and Dyer's TextBook. We have built our custom partitioner class *WordCountPartitioner* to streamline the output of mapper into 3 reducers , one each for words, hashtags and mentions.

  Mapper: TokenizerMapper
  Combiner: In-mapper
  Partitioner: WordCountPartitioner
  Reducer: IntSumReducer

  PseudoCode:

```
class Mapper
        method Initialize
                H<- new AssociativeArray
        method Map(docid a, doc d)
                for all term t ∈ doc d do
                        H{t} ← H{t} + 1
        method Close
                for all term t ∈ H
                        Emit(term t, count H{t})
```

```
class Reducer
    method Reduce(term t, counts [c 1 , c 2 , . . .])
        sum <- 0
        for all count c ∈ counts [c 1 , c 2 , . . .] do
            sum <-sum + c
        Emit(term t, count sum)
```

- The second phase involves secondary sorting using a combined key(word + count). Our partitioner uses the *compareTo()* method of the custom *PairKey* class to sort it into descending order by count.

  Mapper : SorterMapper
  Partitioner : SorterPartitioner
  Reducer : SorterReducer

  PseudoCode:

```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            emit(t.string+t.count,t.count)

class Reducer
    method Reduce(term t, counts [c1 ,c2 , . . .])
        sum <- 0
        for all count c ∈ counts [c1,c2 , . . .] do
            sum <-sum + c
        Emit(term t, count sum)
```

Results : After running the Mapreduce tasks for wordcount on the aggregated data , the following were the top trending results.

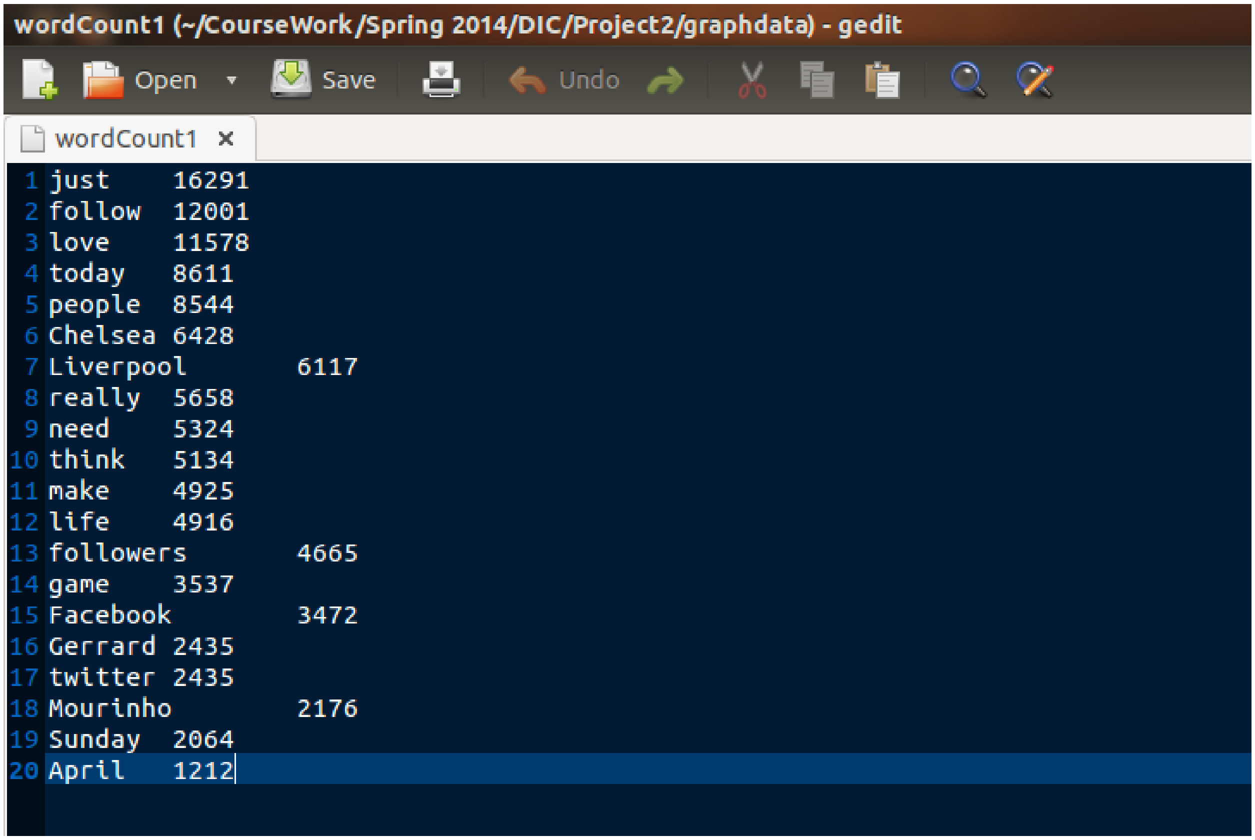


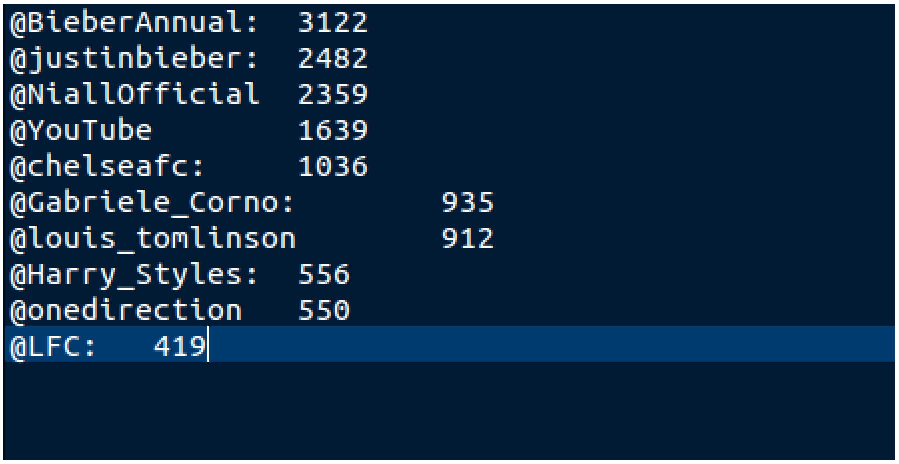
Fig 2. Counts for the top 20 trending words

Fig 3. Tag graph representation for the top 20 trending words

```
#gameinsight	2887
#androidgames,	1478
#CFC	1399
#FilmTANIA	1283
#android,	1238
#LFC	1137
#AdminDirectPopularPENIPU	1029
#ipadgames,	986
#4MusicLFSSwifties	963
#PBBALLINKickOff	917
#openfollow	895
#YNWA	845
#ipad,	828
#4MusicLFSNavy	767
#KangenKakRianaAntoinette	740
#RemajaIndonesia	674
#4MusicLFSBeliebers	668
#CFCLive	664
#PBBScripted	638
#30TwitterosQueQuieroConocer	590
```

Fig 4. Counts for the top 20 trending hashtags

Fig 3. Tag graph representation for the top 20 trending hashtags



@BieberAnnual:	3122
@justinbieber:	2482
@NiallOfficial	2359
@YouTube	1639
@chelseafc:	1036
@Gabriele_Corno:	935
@louis_tomlinson	912
@Harry_Styles:	556
@onedirection	550
@LFC:	419

Fig 5. Counts for the top 10 trending mentions

Fig 6. Tag graph representation for the top 10 trending mentions

2. Word Co-occurence

Requirement Specification:

The task at hand is to find out the co-occurring words in all the tweets that we have collected thus far. The tweets collected were cleaned to remove stop words and unnecessary punctuations to arrive at a clean data set after the required Exploratory data analysis.

When co-occurrence is considered we mainly have two approaches. We can consider all possible pair of words in a single tweet, or we can keep track of occurrences using in associative array. The former approach is rightfully named as Pairs and the latter as Stripes.

The requirement is to generate the relative co-occurrence of words, in both of these approaches.

Implementation Details:

For Stripes approach we scan each word and emit a ([word,neighbour],count) key value pair, one for each of its neighbours in a single tweet. Since this would only give as the regular count we are also emitting a special key pair as discussed in the Text illustrated as ([Word,*],count). This special pair would give us all possible counts of ocurrences of 'Word'. The emitted pairs are then partitioned only on the basis of first key so that the all word pairs including special pair arrive at the same reducer. To ensure that the special pair reaches before regular pair, we make use of compareTo function in the WritableComparable class. Hence at the reducer we can just proceed to divide the regular counts by the marginal or total counts to arrive at the relative frequency for each pair.

Mapper:PairsMapper
Partitioner:PairsPartitioner
Reducer:PairReducer
Driver:PairsDriver

In the case of Stripes, this approach becomes all the more easier. This is because since we are obtaining the result in an associative array or specifically a MapWritable, we can easily calculate the total as well as relative occurance at the reducer.

Mapper:StripesMapper
Reducer:StripesReducer
Driver:StripesDriver

PseudoCode

```
Class Mapper(Object, Text)
{
for each line in Text
Output special pair
for each neighbour
output regular pair,count
}

Class Reducer(PairsKey, IntWritable)
{
for each input key pair
If special then set total count
else
if regular pair then set count/total to get relative count
output the same
}
```

Stripes

Class Mapper(Object, Text)
{
for each line
for each word add neighbours into associative array
output key and MapWritable caontaining neighbours and counts
}

Class Reducer(Text, MapWritable)
{
For each map add the corresponding entries element wise.
For each word in Text get count from all of the entries in Map to get total
Emit key, and each neughbour with count/total count to get relative count.
}

Results:

For the results we have the following screen shots indicating the reducer outputs. We have tested our implementation with 3 reducers and hence have three output files as shown:

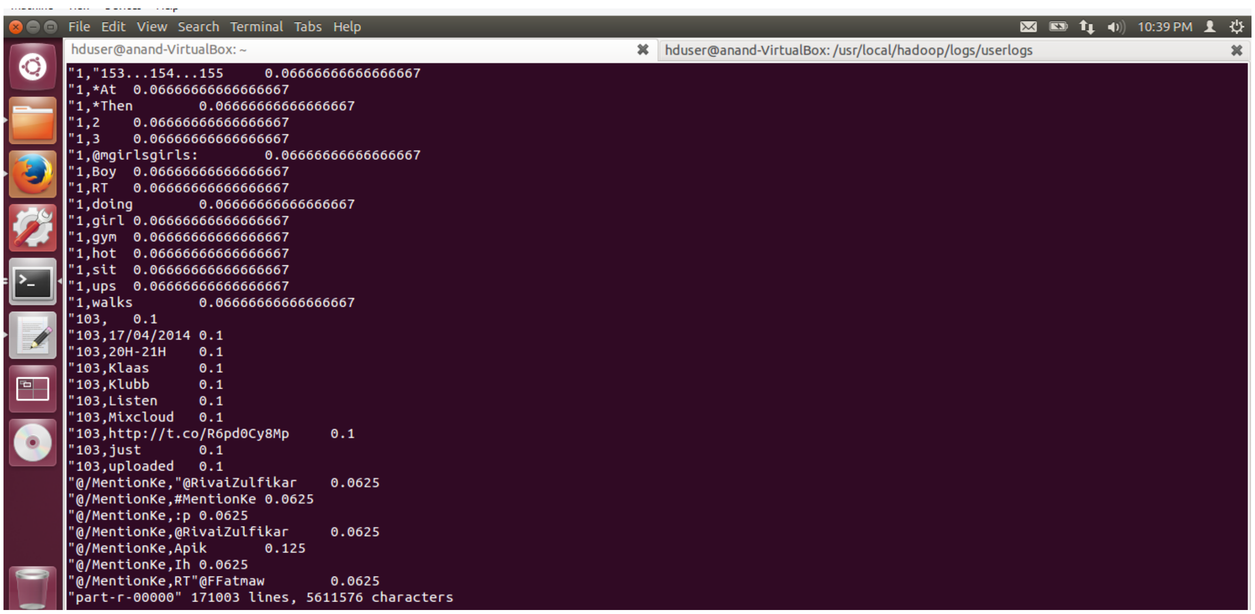


Fig 7:Pairs output: part -r- 00000

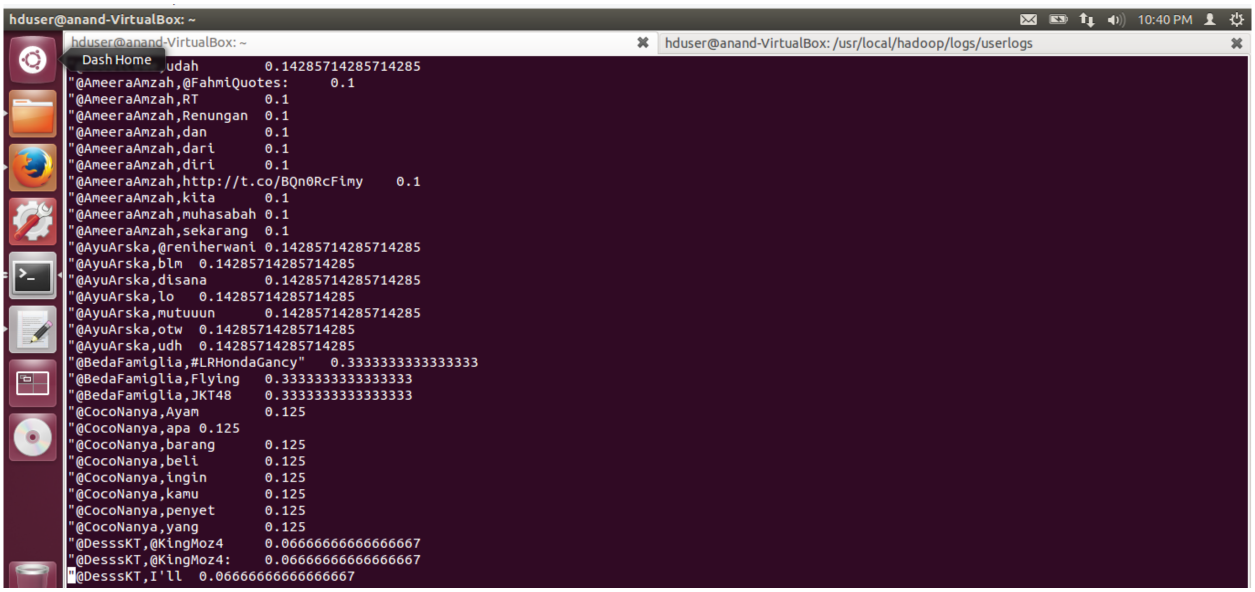


Fig 8: Pairs output:part -r- 00001

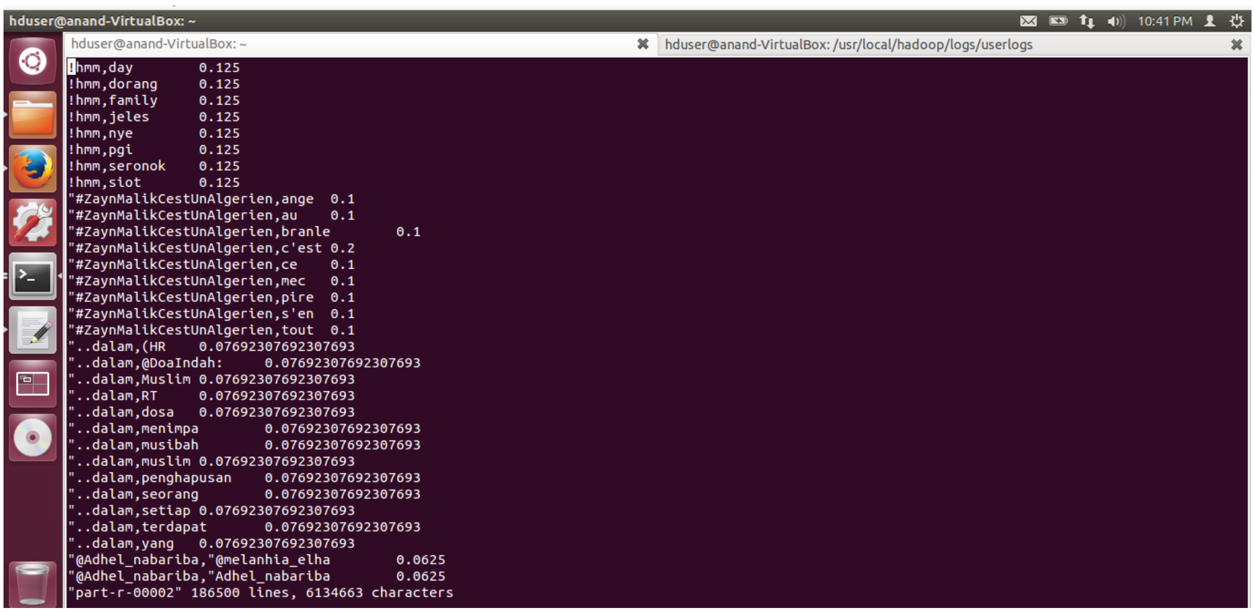


Fig 9 : Pairs output : part-r-00002

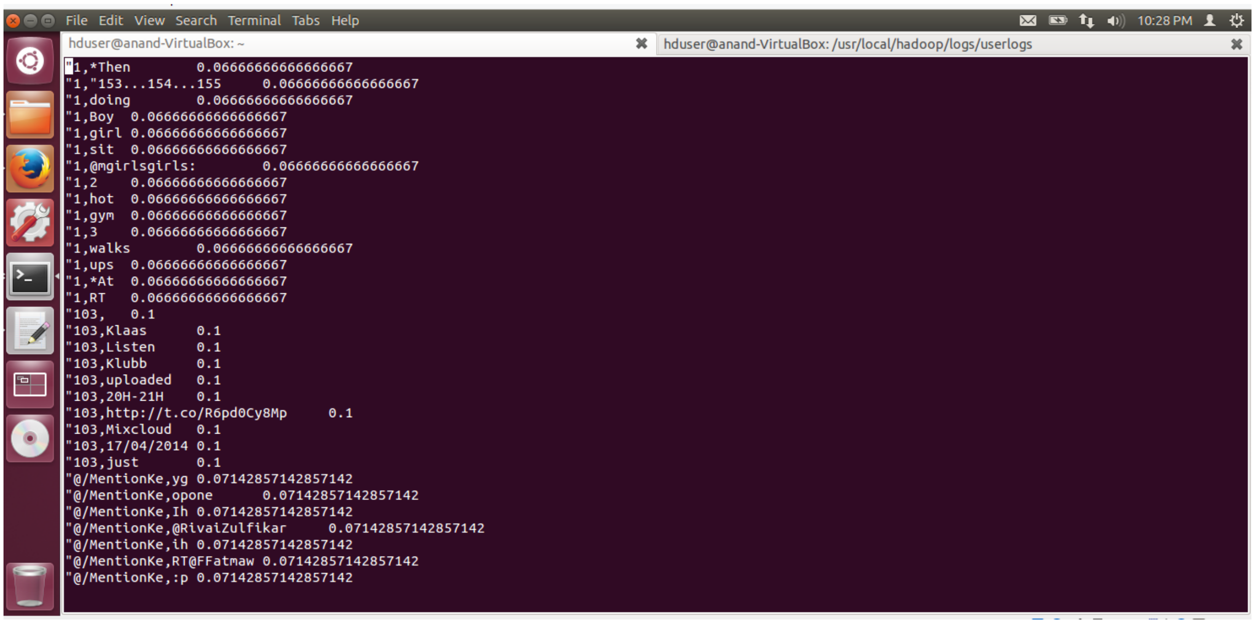


Fig 10 : Stripes output: part-r-00000

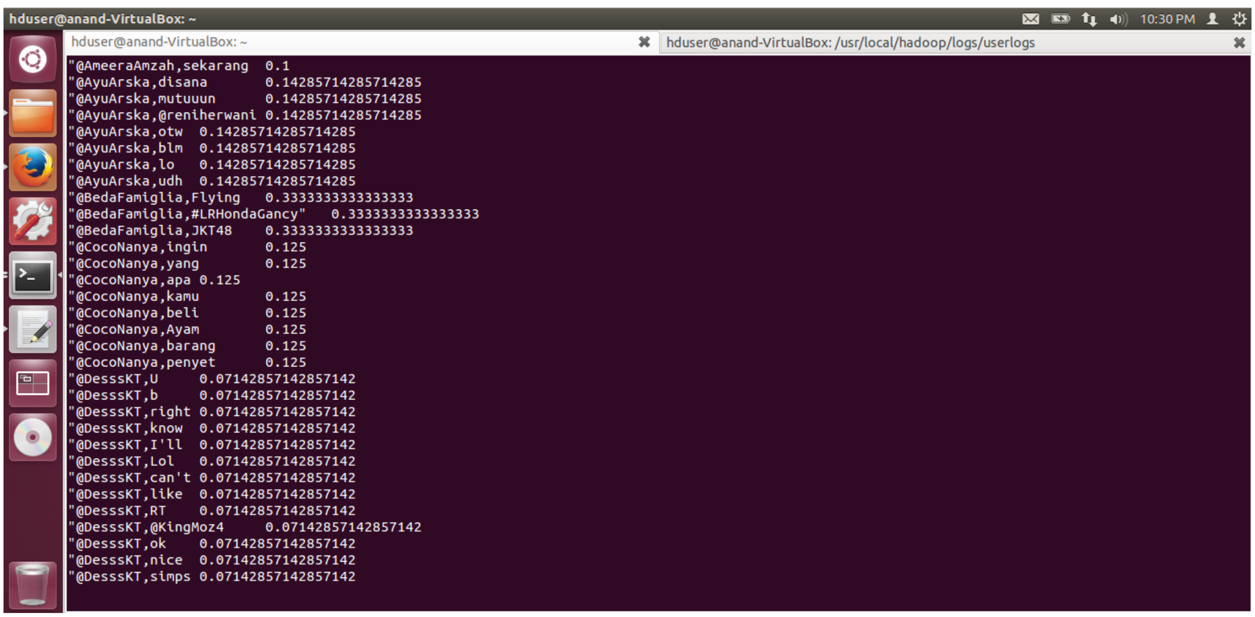


Fig 11 : Stripes output: part-r-00001

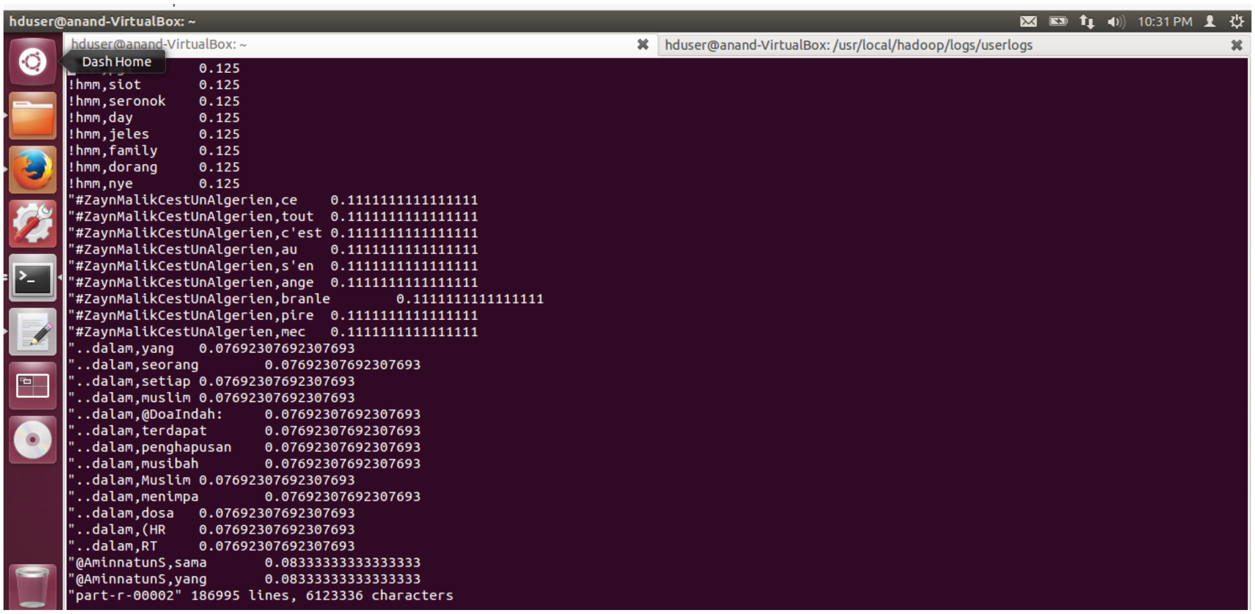


Fig 12 : Stripes output: part-r-00002

3. K-means Clustering

Requirement Specification: We are required to cluster the tweeters based on the number of followers they have, into three sets . In a real world scenario, this information may be used by advertising agencies.

Implementation details: We fulfil this task by implementing a mapreduced version of the K-means clustering algorithm. We use multiple iterations of MR to arrive at a converged value for chosen centroids. We make use of counters to save the state in between iterations.

Mapper: PrimaryMapper/SecondaryMapper(for further iterations)
Reducer: ClusterReducer
Combiner: ClusterReducer

Partitioner: None (Single Reducer)

PseudoCode:

```
class Mapper
        method Initialize()
                CentroidList=readFromFile("CentroidsPath")
        method Map(docid a, doc d)
                for all term t ∈ doc d do
                        minDistance=Inf.
                        for(c :  CentroidList)
                                distance=abs(t.followerCount-c)
                                if(distance<minDistance)
                                        minDistance=distance
                                        nearestCentroid=c

                        emit(nearestCentroid,t)



class Reducer
        method Initalize()
                Counter=0
                newCentroids=new List()
        method Reduce(Centroid c, Terms[t1,t2….])
                sum=0
                count=0
                for(t: Terms)
                        sum=sum+t.followerCount
                        count=count+1;
                        emit(c,t)

                avg=sum/count
                if(avg!=c)
                        Counter=Counter+1
```

```
        newCentroids.add(avg)


method Close()
        newCentroids.writeToFile("CentroidsPath")
```

<u>Result:</u> We took our initial centroids at 100, 2000 and 10000 and the final centroids for the clusters (after 21 iterations) were 1522, 58697 and 3645047

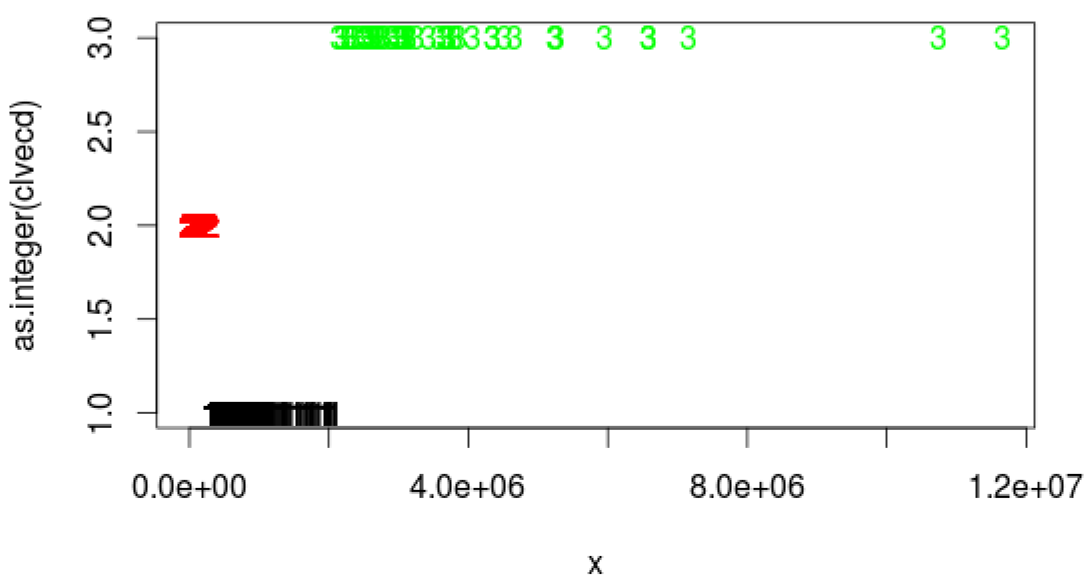


Fig 13 : Clusters plotted using R

4. Shortest Path in Graph

The task at hand is to implement the MapReduce version of the Dijkstra's Algorithm for shortest paths to all nodes of the given graph from a single source node. Here we have a connected directed graph as input with adjacency list indicating neigbours and distance stored alongside. We map over all nodes. Mappers emit a key-value pair for each neighbor on the node's adjacency list. Where the key value is the node id of the neighbor and value is current distance to node +1. The logic behind this is if we can reach node n with d distance then all connected nodes of n can be reached with d+1. After shuffle reducers have keys corresponding to nodes and distances about all path leading to the node. The reducer then selects shortest and updates the node.

```
class Mapper
{
map(nid, node N)
d=N.distance
emit(nid ,N)
for all node m adjacent to N
emit(m,d+1)
}

class reducer
{
reduce(nid m,[d1,d2..])
dmin=infinity
M=null;
for all d in d1,d2..
if d is a node then
M=d
```

else if d<dmin
dmin=d
m.distance = dmin
emit(m,M)
}

Results:

```
1 0 2:
2 1 3:
3 2 2:13:
4 5 14:
5 5 14:
6 10 7:
7 9 6:8:17:
8 10 7:20:
9 13 10:
10 12 9:20:
11 5 12:
12 4 11:13:22:
13 3 3:12:14:
14 4 4:5:13:15:
15 5 14:16:
16 6 15:27:
17 8 7:27:
18 13 19:35:
19 12 18:20:21:
20 11 8:10:19:
21 13 19:
22 5 12:24:
23 8 32:
24 6 22:32:
25 8 26:32:
26 8 25:27:
27 7 16:17:26:28:33:
28 8 27:44:
29 17 37:38:
30 16 45:
31 8 32:
32 7 23:24:25:31:40:41:46:
33 8 27:42:43:
34 17 49:
35 14 18:50:
36 16 50:
```

Fig 15 : Subset of the output of the output of the Dijkstra's Algorithm

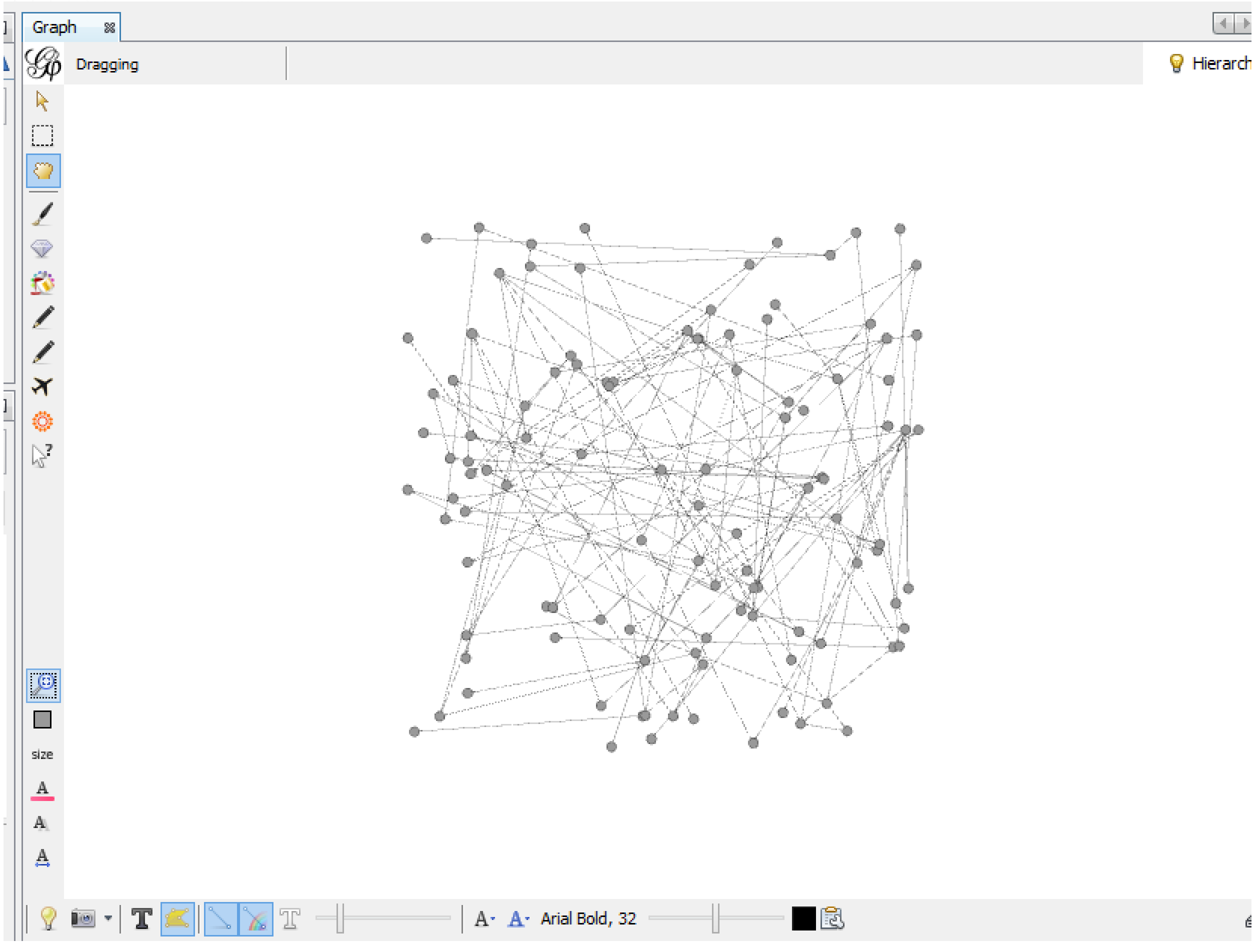


Fig 16 : Visual representation of the given graph (using Gephi)