BIG-DATA CONTENT RETRIEVAL, STORAGE AND ANALYSIS

CSE 587 - DATA INTENSIVE COMPUTING

UNIVERSITY AT BUFFALO

Authors:
HARISH MANGALAMPALLI
hmanagala@buffalo.edu
RAJARAM RABINDRANATH
rajaramr@buffalo.edu

Contents

DATA AGGREGATOR

We had chosen Twitter4j a java implementation for Twitter API to help fetch tweets from the twitter stream. We had chosen to filter in tweets that were relevant to the INDIAN GENERAL ELECTIONS 2014. In order to make sure that we did not bias our collection of tweets towards the Elections we had included filter words that were not necessarily strongly tied up with that said topic. The filter words used on the stream were:

- India, 2014, Elections2014,IndiaVotes among others

Some details about the data that was collected:

- Data Format: text files
- Volume of data: Data was collected non-stop for 9 days
    - Total number of tweets : 12,982,122 tweets
    - Since the data is too massive for any single machine to handle (please Infrastructure section) we decide to have the code run on data collected in a single day
    - We chose Apr 17 : 1,552,886 tweets

We present our results in the following section

INFRASTRUCTURE

Single node Hadoop clusters: One/team member

- Rajaram Rabindranath: 200 GB / 4GB RAM
- Harish Mangalampalli: 1TB / 8GB RAM

MAP-REDUCE WORK-FLOWS

Simple word-Count

Objective:

Compute the word count of all unique words in the tweet corpus, post filtering out stop words.

Process:

Use map-reduce framework to split the task at hand into smaller manageable workloads and then compute word count for each unique word and aggregate the same using a reducer.

Trends

Objective:

Find the words that have the highest occurrence count in the tweets that are available in the tweet corpus.

Process:

This requirement is an extension of word count, wherein post word count one chooses the top words in terms of occurrence.

#tag counts

Objective:

Find the count of unique has tags occurring in the tweet corpus

Process:

This requirement again is an extension of word count, however in this case one shall be looking for special words; ones that start with a #tag. We then proceed to find count of occurrence of these special words.

@xyz

Objective:

Find the count of unique user tags in all the tweets available in the tweet corpus

Process:

This requirement again is an extension of word count wherein we look for special words in tweets i.e. the ones that start with an @ symbol indicating a user tag.

Pseudo Code for all the above objectives:

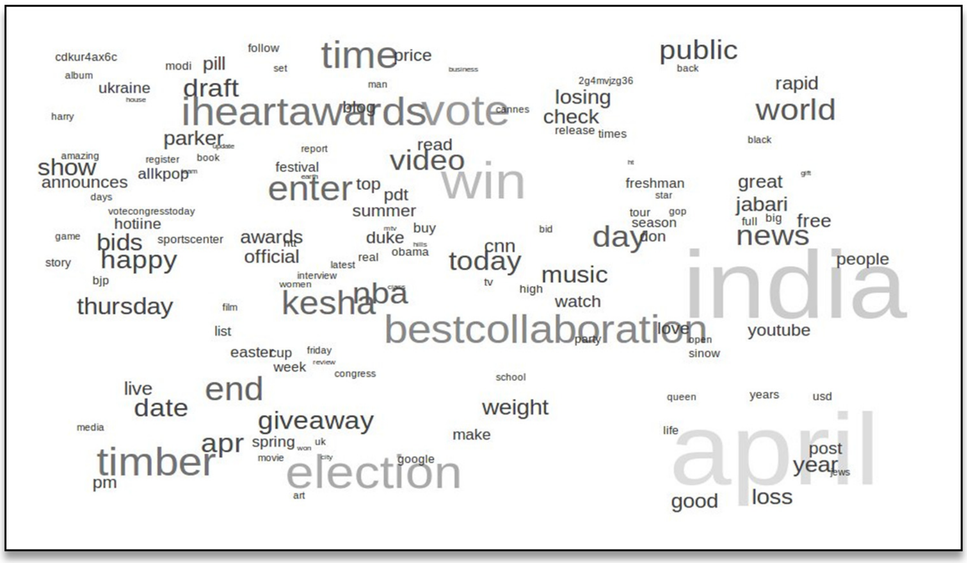
```
class Mapper
{
      method Map(docid a, doc d)
      {
            for all term t belongs to ‘doc d’ do
                  Emit(term t, count 1);
      }
}

class Reducer
{
      method Reduce(term t, counts [c1, c2, ...])
      {
            sum ← 0;
            for all count c belongs to counts [c1, c2, ...] do
            {
                  sum ← sum + c;
                  Emit(term t, count sum);
            }
      }
```

}

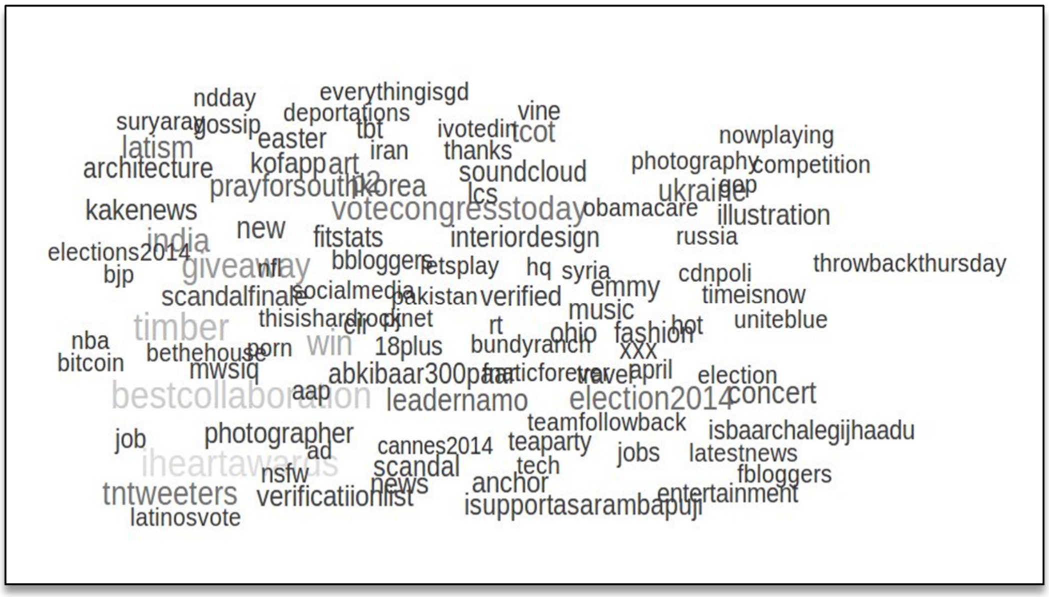
Results:

Following are the results of the above problems. We have image clouds for each of the objectives listed above. As we had collected twitter data by filtering in those tweets that had either IndianElection/India/2014/Election2014 of these terms present in them, it is evident in the visualization that reasonable number of tweets collected have been about the Indian elections.

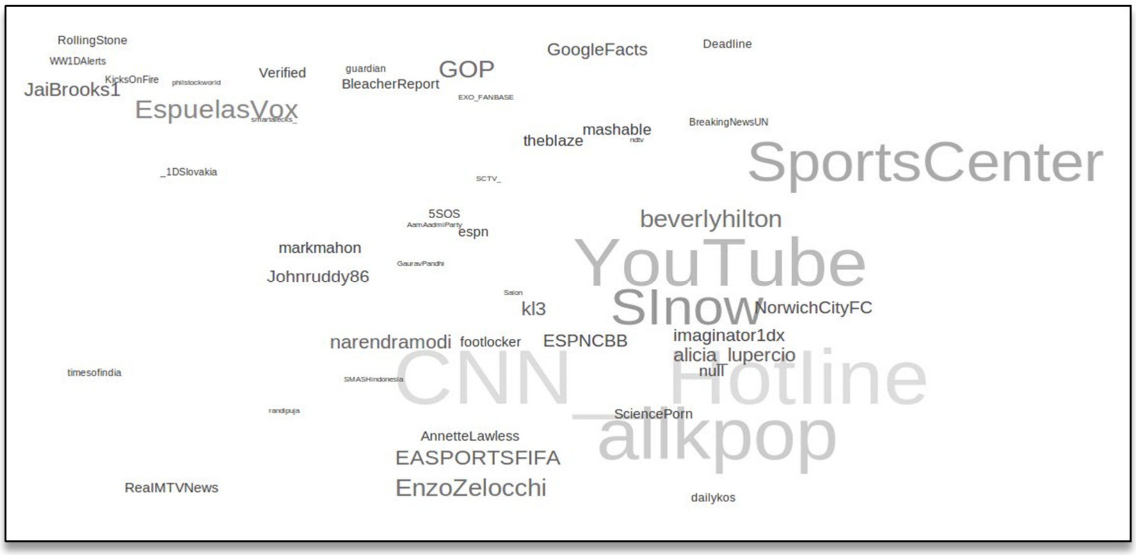
WORD CLOUD IMAGE



HASHTAG CLOUD IMAGE



USER TAGS (@xyz)



PAIRS & STRIPES APPROACH – #TAG CO-OCCURRENCE

Pairs

Objective:

Find co-occurrence of hash tags using pairs approach

Process:

Given a set a hashtags in the tweet corpus that occur together in a tweet we need to compute the number of occurrences of each perm/combination of the above mentioned hashtags

Pseudo Code:

```
class Mapper
{
      method Map(docid a, doc d)
      {
            for all term w belongs to doc d do
            {
                  for all term u belongs to Neighbors(w) do
                  {Emit(pair (w, u), count 1);}
            }
      }
}
```

```
class Reducer
{
      method Reduce(pair p, counts [c1, c2, ...])
      {
            s ← 0;
            for all count c in counts [c1, c2, ...] do
            {s ← s + c;} // sum co-occur count
            Emit(pair p, count s);
      }
}
```

Results:

Please look for pairs.txt in the “output” folder

Stripes

Objective:

Same as above but using the stripes method wherein we compute the marginal of occurrence

Process:

Pseudo Code:

```
class Mapper
{
      method Map(docid a, doc d)
      {
            for all term w belongs to doc d do
                  H ← new AssociativeArray
            for all term u belongs to Neighbors(w) do
            {H{u} ← H{u} + 1}//Tally words co-occurring with w
            Emit(Term w, Stripe H)
}
class Reducer
{
      method Reduce(term w; stripes [H1,H2,H3,...])
      {
            $H_f$ <-  new AssociativeArray
            for all stripe H belongs to stripes [H1,H2,H3,...] do
            {Sum($H_f$, H)}//Element-wise sum
            Emit(term w, stripe $H_f$ )
      }
}
```

Results:

Please look for stripes.txt in the “output” folder

PAIRS & STRIPES APPROACH – RELATIVE #TAG CO-OCCURRENCE

Stripes

Pseudo code:

```
class Mapper
{
      method Map(docid a, doc d)
      {
            for all term w belongs to doc d do
                  H ← new AssociativeArray
            for all term u belongs to Neighbors(w) do
            {
                  H{u} ← H{u} + 1
                  H{*} ← H{*} + 1
            }//Tally words co-occurring with w
            Emit(Term w, Stripe H)
}
class Reducer
{
      marginal ← 0;
      method Reduce(term w; stripes [H1,H2,H3,...])
      {
            $H_f$ ← new AssociativeArray
            for all stripe H belongs to stripes [H1,H2,H3,...] do
            {
                  if(H is *)
                        marginal ← Sum(H)
                  else
                        Sum($H_f$,H)/marginal
            }//Element-wise sum
            Emit(term w, stripe $H_f$ )
      }
}
```

Result

Please look for stripes_relative.txt in the “output” folder

Pairs

Pseudo code:

```
class Mapper
{
      method Map(docid a, doc d)
      {
            for all term w belongs to doc d do
            {
                  for all term u belongs to Neighbors(w) do
                        Emit(pair (w, u), count 1);
                        Emit(pair (w, *), count 1);
            }
      }
}

class Reducer
{
      marginal ← 0;
      method Reduce(pair p, counts [c1, c2, ...])
      {
            if(Neighbour(p.w) is *)
            {
                  for all count c in counts [c1, c2, ...] do
                  {
                        marginal ← marginal + c;//Sum co-occurrence counts
                  }

            }
            s ← 0;
            for all count c in counts [c1, c2, ...] do
            {
                  s ← s + c;//Sum co-occurrence counts
            }
            Emit(pair p, s/marginal);

      }
}
```

Result:

Please look for pairs_relative.txt in the “output” folder

CLUSTERING & GRAPH ALGORITHMS

K-Means (ITERATIVE MAP-REDUCE)

Objective:

Design an iterative map-reduce code to perform k-means on a given dataset. The use of counters is key to this implementation both from the design and marks point of view.

Process:

Having collected the twitter data we proceed to cluster users based on three attributes; namely

- **Friends' Count**
- **Followers' Count**
- **Status Count**

These three features together can be construed as "Influence of a user". Since we do not know the natural distribution of data and the clustering of the users based on the aforementioned derived metric we perform K-means on our dataset to ascertain the clustering. Since K-means requires the K cluster centers to stabilize the algorithm is iterative in nature and has to be adapted to map-reduce framework and this is done using iterative map-reduce which in turn is accomplished via the use of counters.

We run the map-reduce version of k-means for different values to K to find the right K, using the with-in cluster sum of squared errors; henceforth referred to as simply SSE.

Counter being used: moreIteration

Pseudo Code:

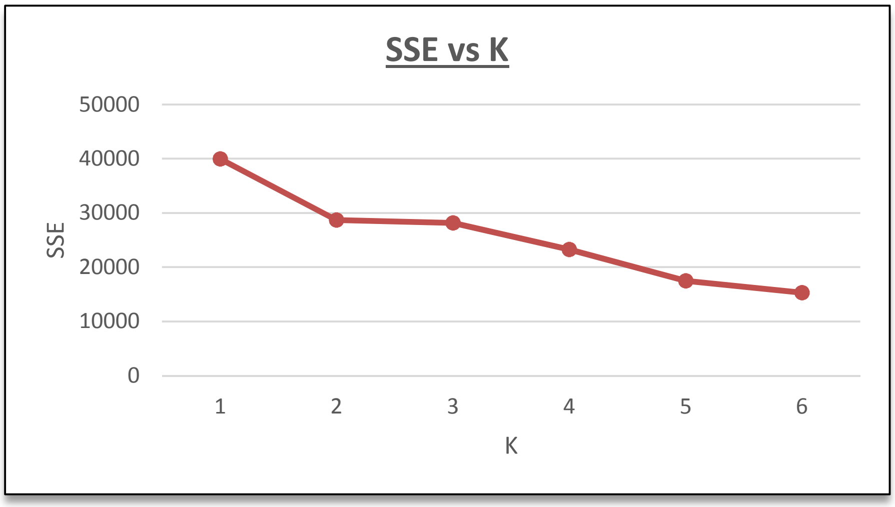
```
class Mapper
{
      List centroids[k]; // read from a file
      List dist[k];
      method Map(docid a,doc d)
      {
            for all line in doc d do
            {
                  create dataPoint: friends_count, followers_count &
status_count
                  for each centroid in centroids
                  { dist[i]=Euclidian( datapoint,centroid)}
                  index=min_index(dist);
                  Emit(index,datapoint)
            }
```

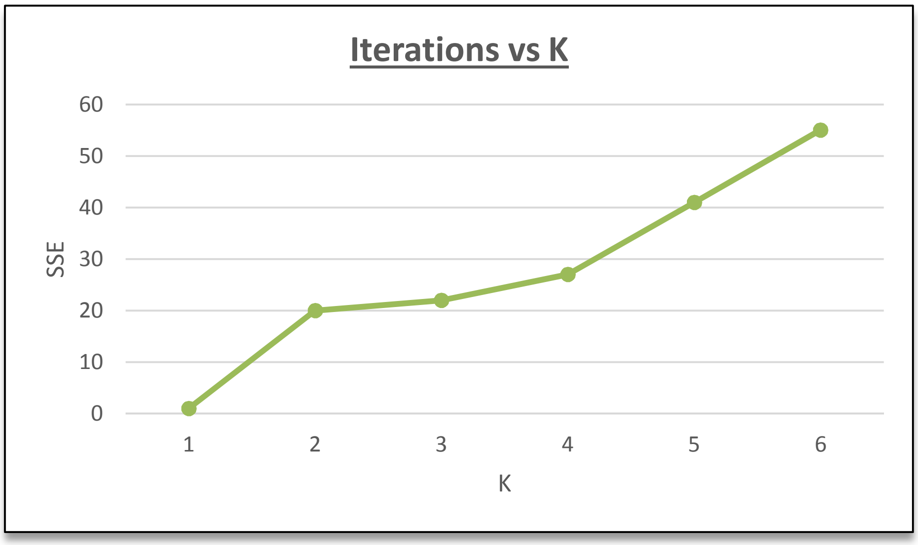
```
        }
}
class Reducer
{
        new centroids[K];
        method Reduce(centroid_id,[dataPoint1,dataPoint2 …..])
        {
                count ← 0;
                for all dataPoint in [dataPoint1 ….. dataPointN]
                {
                        centroid[centroid_id].friends +=dataPoint.friends;
                        centroid[centroid_id].followers +=dataPoint.followers;
                        centroid[centroid_id].status +=dataPoint.status;
                }
                centroid[centroid_id].friend =centroid[centroid_id].friend/count;
                centroid[centroid_id].followers=centroid[centroid_id].followers/c
        ount;
                centroid[centroid_id].status =centroid[centroid_id].status/count;
        }
        method cleanup()
        {
                //check if the centroids have converged
                // if not one more iteration required
                // else exit without more iterations
        }
}
```

<u>Results:</u>

The following results that display the SSE corresponding to the value of K we experimented with and using the *elbow-rule* (evident from the graph SSE vs K) we choose K=2 as the right number of K for clustering the user in our data.

K	SSE	ITERATIONS
1	39958.51972	1
2	28705.54372	20
3	28159.9249	22
4	23275.86302	27
5	17480.06623	41
6	15322.75924	55

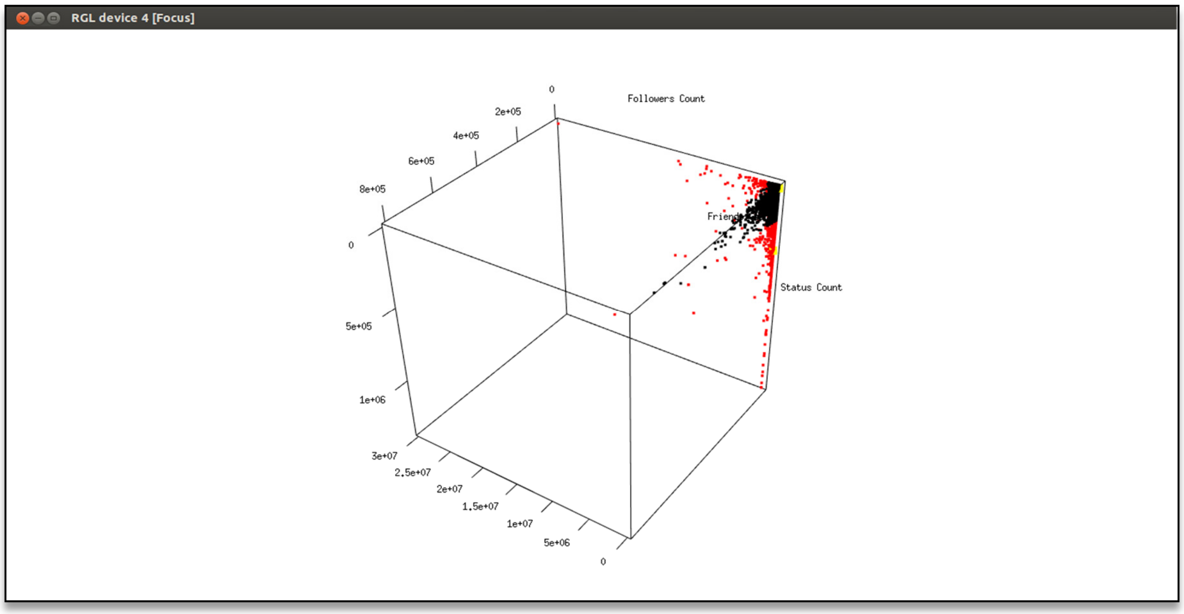


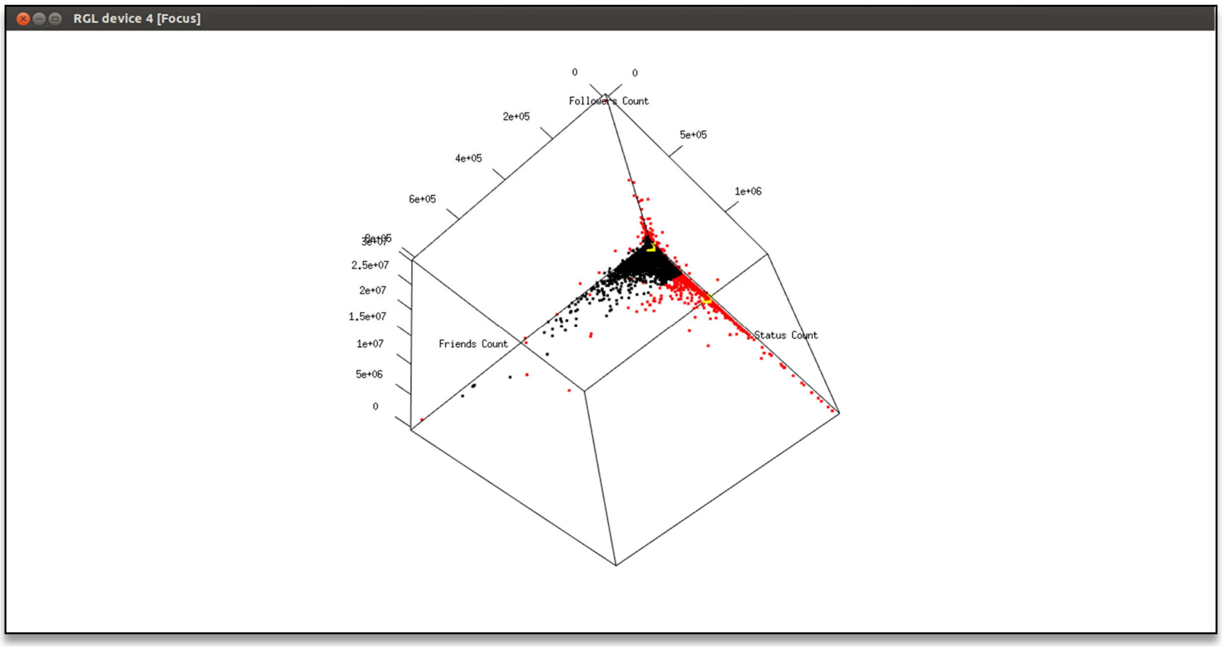


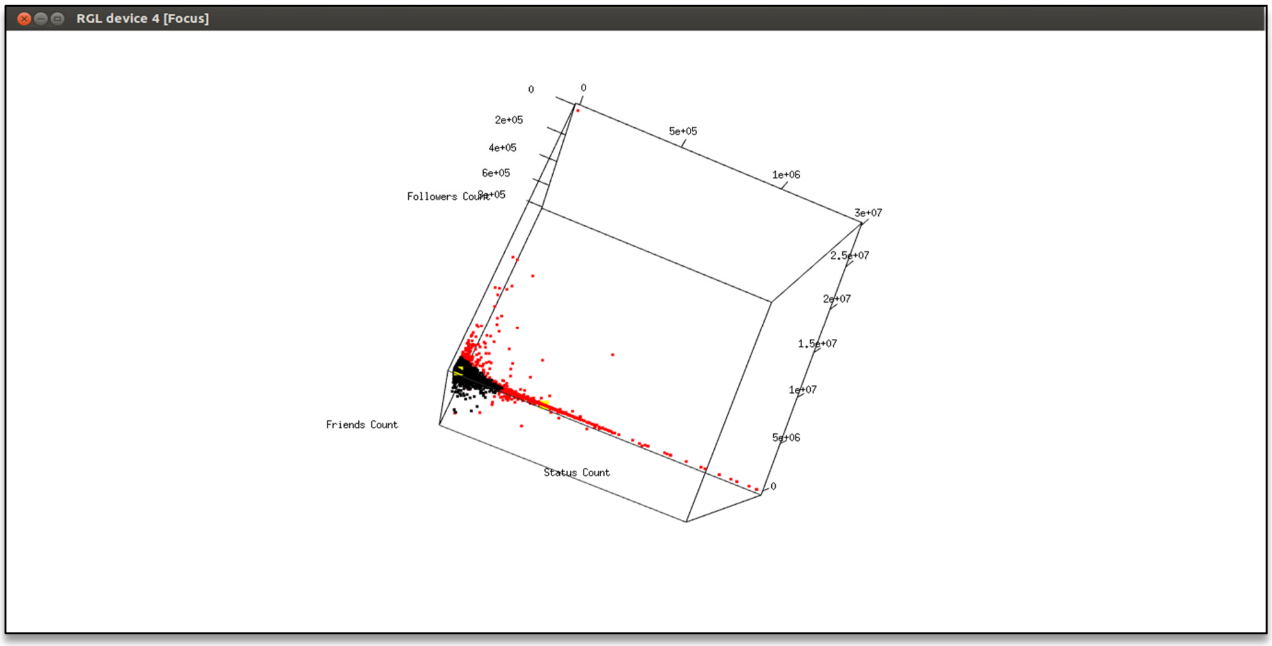
FOLLOWING IS THE CLUSTER CENTERS FOR K=2

clstID	friend	follower	status	member count
0	1122.679	2783.183	18167.77	1509828
1	2763.717	32295.67	406467.4	43058

VISUALIZATIONS OF CLUSTERS & CENTROIDS K=2[Centroids are marked in yellow]


RGL device 4 [Focus]
Followers Count
Friends Count
Status Count
0
2e+05
4e+05
6e+05
8e+05
0
5e+05
1e+06
3e+07
2.5e+07
2e+07
1.5e+07
1e+07
5e+06
0


RGL device 4 [Focus]
0
0
Followers Count
2e+05
4e+05
6e+05
8e+05
5e+05
1e+06
3e+07
2.5e+07
2e+07
1.5e+07
1e+07
5e+06
0
Friends Count
Status Count



Graph Algorithm (ITERATIVE MAP_REDUCE)

Objective:

Implement single source shortest path using iterative map-reduce to find to compute the distance of the source node from every other node on corpus data that we have collected.

Process:

We selected the"***@UBCommunity"*** members as our data set for our study on shortest path. We collected the followers of this community and further expanded the set of connections by collecting the followers of each member in the community. The idea behind the collection pattern was to find out the connectivity between members of the UB Community in the absence of this node in the graph so as to study how members of the community are inter-connected.

We also ran our shortest path code on the "large" graph dataset given to us; we then proceeded to visualize the same using sigma.js, please refer the results section for the visualization.

Pseudo Code:

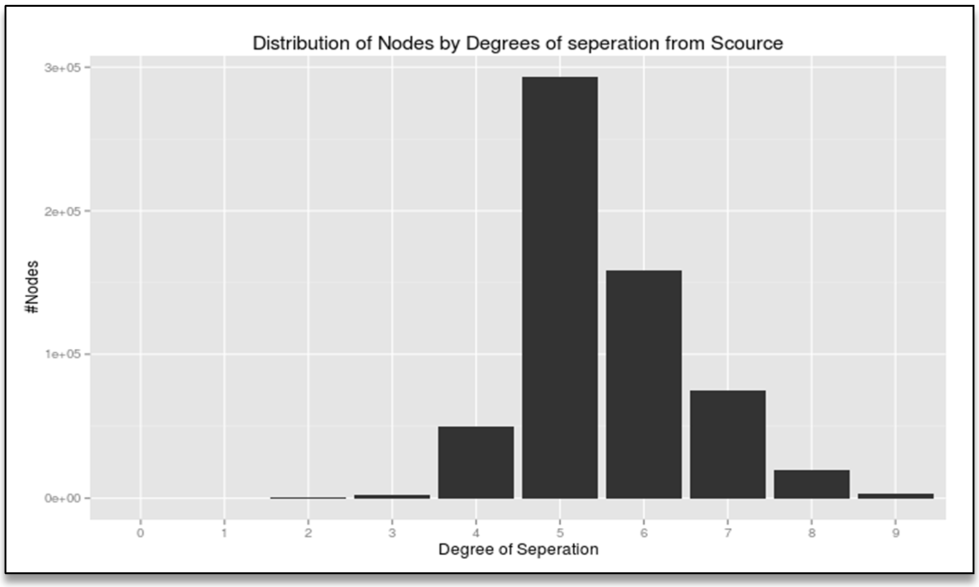
```
class Mapper
{
      method Map(nid n, node N)
      {
            d ← N.Distance
            Emit(nid n, N) //Pass along graph structure
            for all nodeid m belongs to N.AdjacencyList do
```

```
                Emit(nid m, d + 1)// Emit distances to reachable nodes
        }
}

class Reducer
{
        method Reduce(nid m, [d1, d2, ...])
        {
                dmin ← infinity;
                M ← null;
                for all d belongs to counts [d1, d2, ...] do
                {
                        if IsNode(d) then
                                M ← d //Recover graph structure
                        else If d < dmin then //Look for shorter distance
                        {
                                dmin ←  d
                                M.Distance ←  dmin //Update shortest distance
                                Emit(nid m, node M)
                        }
                }
        }
}
```

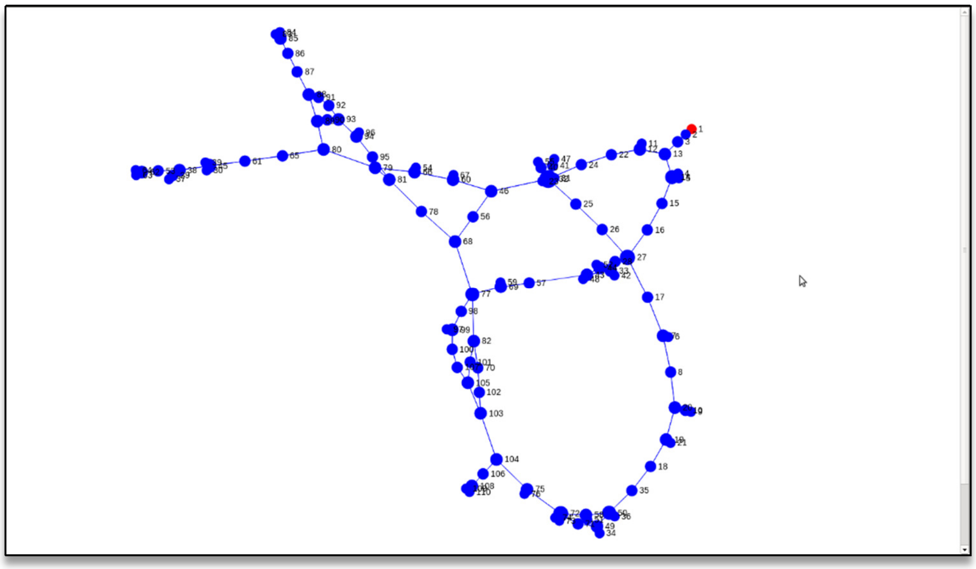
Results:

Result of the algorithms execution. Given the large value of data we list the count of nodes for a given degree of separation. The following table list the degrees of separation and the number of nodes in the data corpus that have the said degree of separation vis-à-vis the source node; followed by a bar graph of the same. Please note that for 1204 nodes the degrees of separation from source node is infinity (i.e. no connection path exist); this is indicated by a "NULL" in the degrees column.

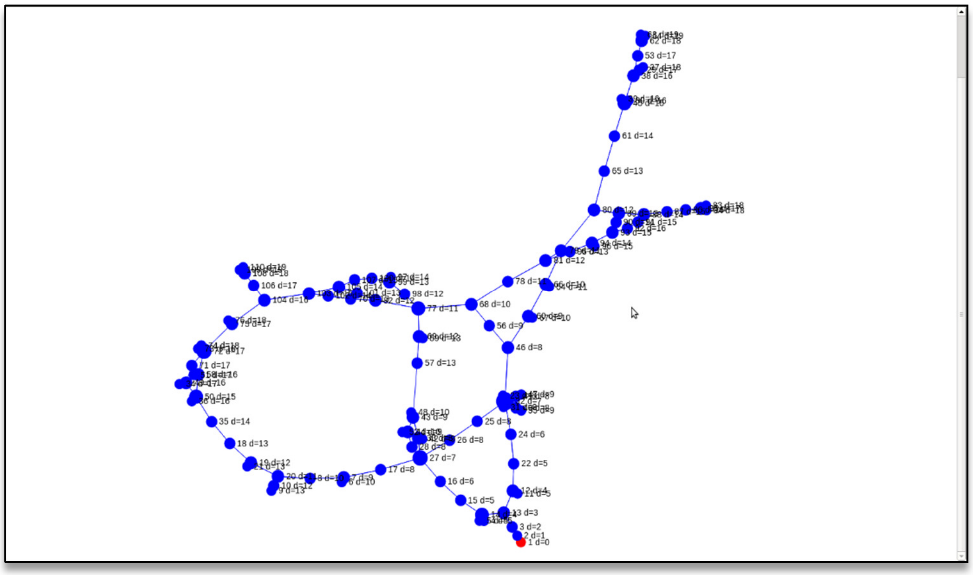
Degrees	#Nodes
0	1
1	40
2	373
3	2004
4	49994
5	293472
6	158988
7	75241
8	19834
9	3633
NULL	1204



THE INPUT GRAPH DATASET



THE OUTPUT GRAPH DATASET - LABELLED



REFERENCES

- LYNN & DYER