Kmeans Clustering with Spark

Motivation

We all know that clustering has many useful applications now such as finding a group of consumers with common preferences, grouping documents based on the similarity of their contents, or finding spatial clusters of customers to improve logistics.

In this project, we will use the k-means algorithm to find the cluster of the locations data. And then we will use our kmeans code to solve other clustering problems.

Because the amount of data is so big and the computation is big too, we decide to uses SPARK to implement our code in an efficient distributed fashion.

Documentation of approach

Why K-Means?

We all know that clustering is the most frequently used way to find the patterns in the data. And k-means algorithm often uses the position or distance of data points from one another to cluster. And the measurements of distance we use are Euclidean distance and Great-circle distance, because our data is in a sphere and great-circle distance can explain the data in the sphere while. Therefore, Great-circle distance is also part of our implementation. And because Spark is good at dealing with iterative and interactive data sets, we use Spark to implement.

The algorithm of K-Means

Given data set (x1,y1), (x2,y2).. (xn,yn), we choose k points from the data set as initial centroids. It is an iterative algorithm. In each iteration:
-Iterate each data point and calulate the distance to each of the k centroids, and assign each them to its closest centroid.

-Then calculate the center of each centroid's members(a cluster), make this center location the new centroid.
So after each iteration, the centroid's location will be updated a little closer to the correct location. And after certain number of iterations, the result will "settle", thus the stopping critaria is that the distance between old and new centroid location is smaller than a threashold.

Implementation with spark

Firstly, pre-process the given data, and we will get (longitude, latitude) pairs. Then we randomly take k samples from the data to form initial centroids. Besides, we transform that centroids as a RDD since sample is a list. Moreover, we use zipWithIndex to create key-value pair (index, location). This rdd will be joined subsequently with the new centroids to compute their distances.

After the pre-processing is done, we form a while loop to run the k-means algorithm, we set the while statement as "not done". "Done" is a flag of whether the algorithm is done calculation. Firstly, we call closestPoint function which returns the index in the array of centroids to the given point to retrieve a rdd of key-value pair(index of the centroids, point location). Besides, the distance measure parameter will decide Euclidean distance or Great Circle distance to use. Then we call rdd.countByKey to get the numbers of the values(point location) that are appended to each centroid. After that, we can use addPoints and the numbers we just computed to retrieve an updated centroids rdd. Then we use join to combine old centroids with the new updated centroids(index,(new centroid location, old centroid location)). Now we can compute the distance of the shifting between each old and new centroid and compare it with threshold value convergeDist. We create a for loop to scan if every changeing distance of the centroids meet the requirement, then the while loop is terminated and k-means calculation is completed.

Data prep and approach

-We use spark to pre-process data, similar to milestone 2, in order to get (lattitude, longitude) pairs.
-also we did handle abnormal data point(ie check length of tuple, filter u'', casting unicode to float for calculation etc)

In terms of approach,
-first run in local cluster
-first run with a subset of data with following command

```
pointSamples = points.sample(False, 20)    #sample will return rdd
```

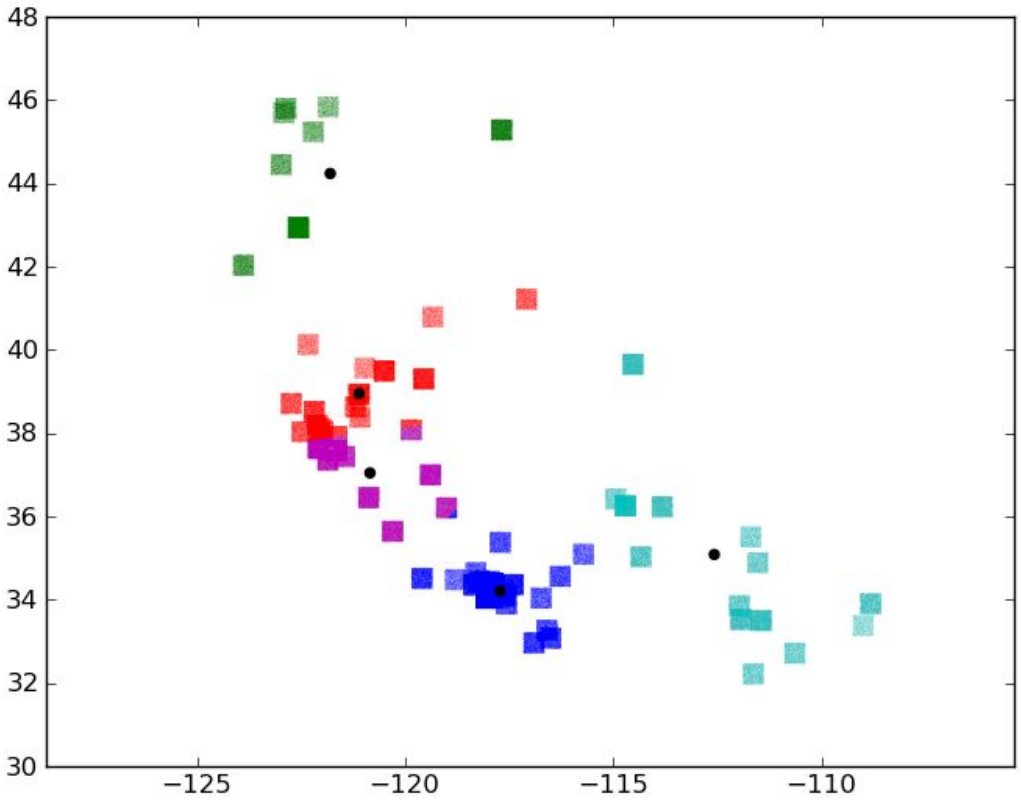
-then run with the whole set and visualization
-then run in cluster mode

Visualization + Result

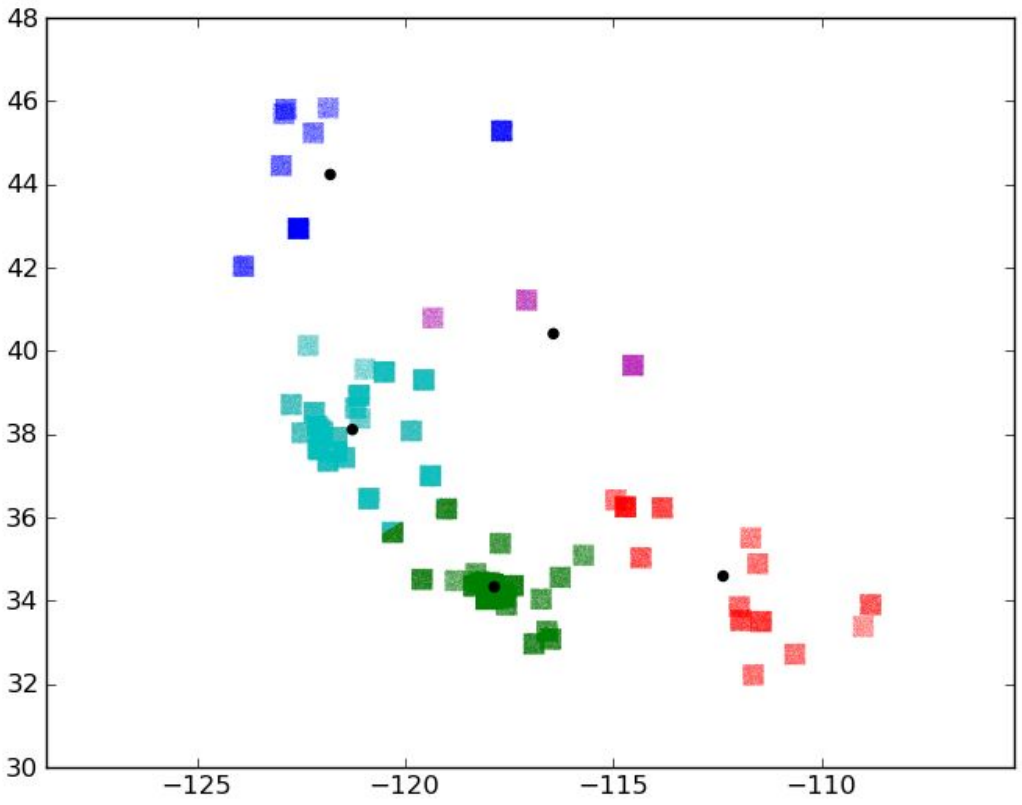
We used our kmeans script on multiple datasets with different input parameters. Results, visualizations are discussed below

Device data

K = 5 Euclidean:



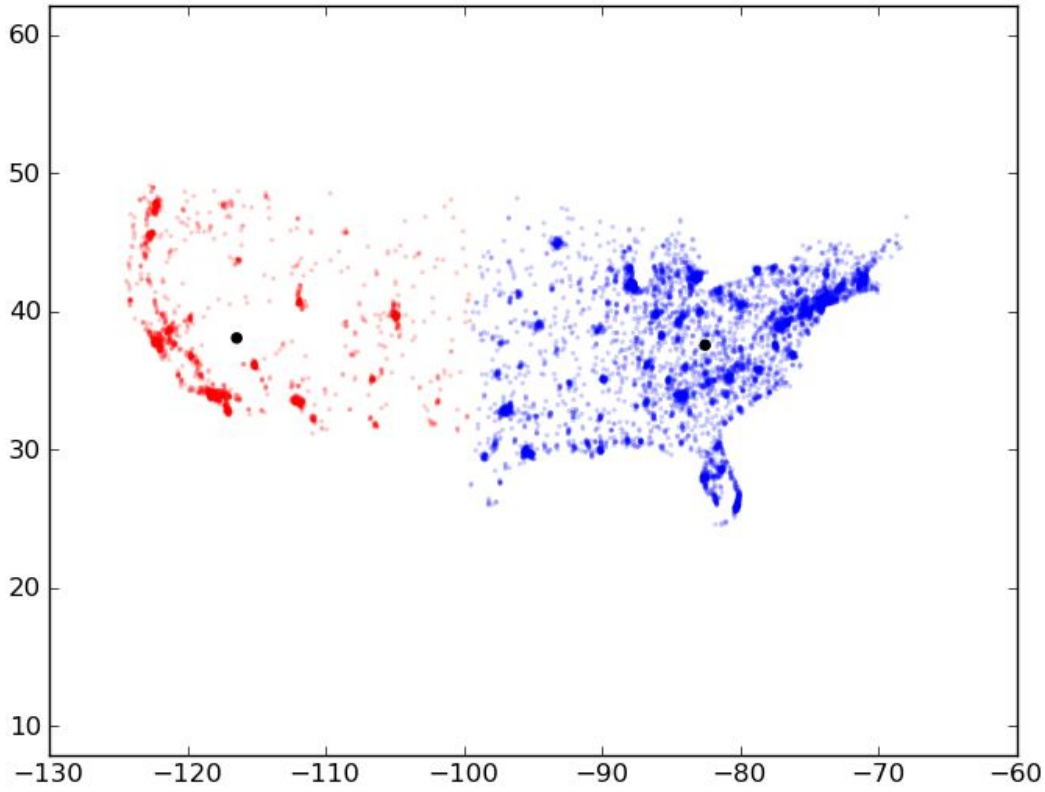
K = 5 GreatCircle:



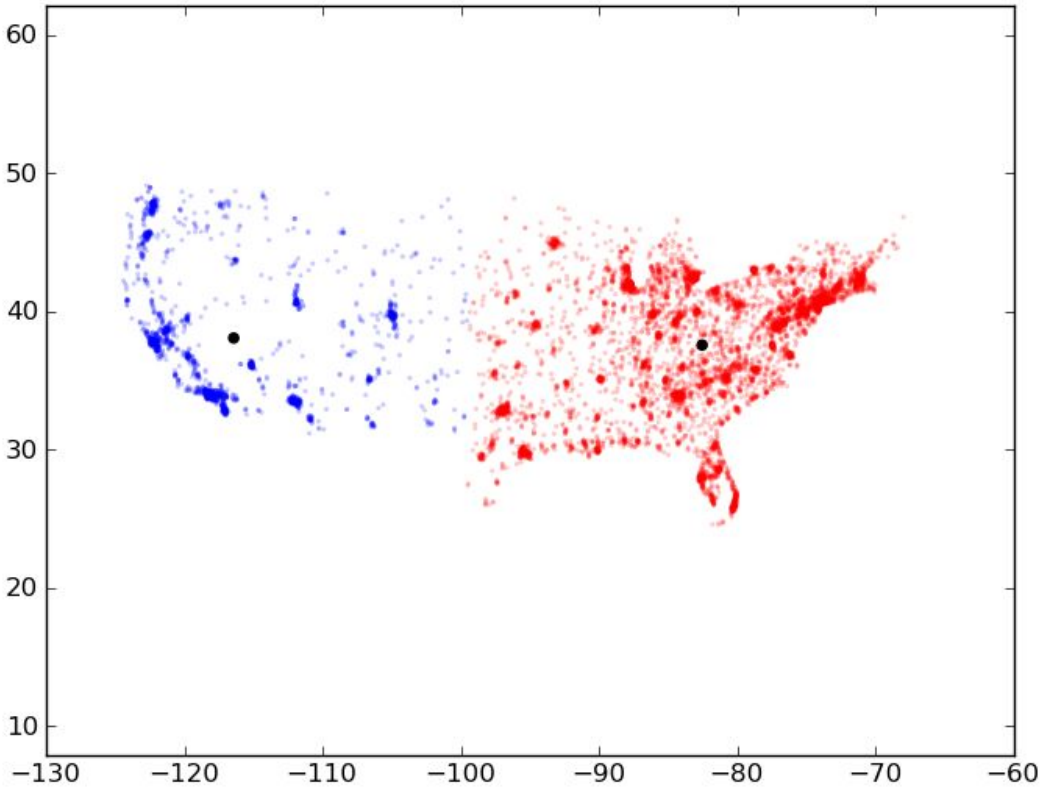
Results/discussion

There is some difference. Euclidean measure seems to distribute points to each cluster more evenly than Great Circle measure.

Synthetic data
K = 2 Euclidean:



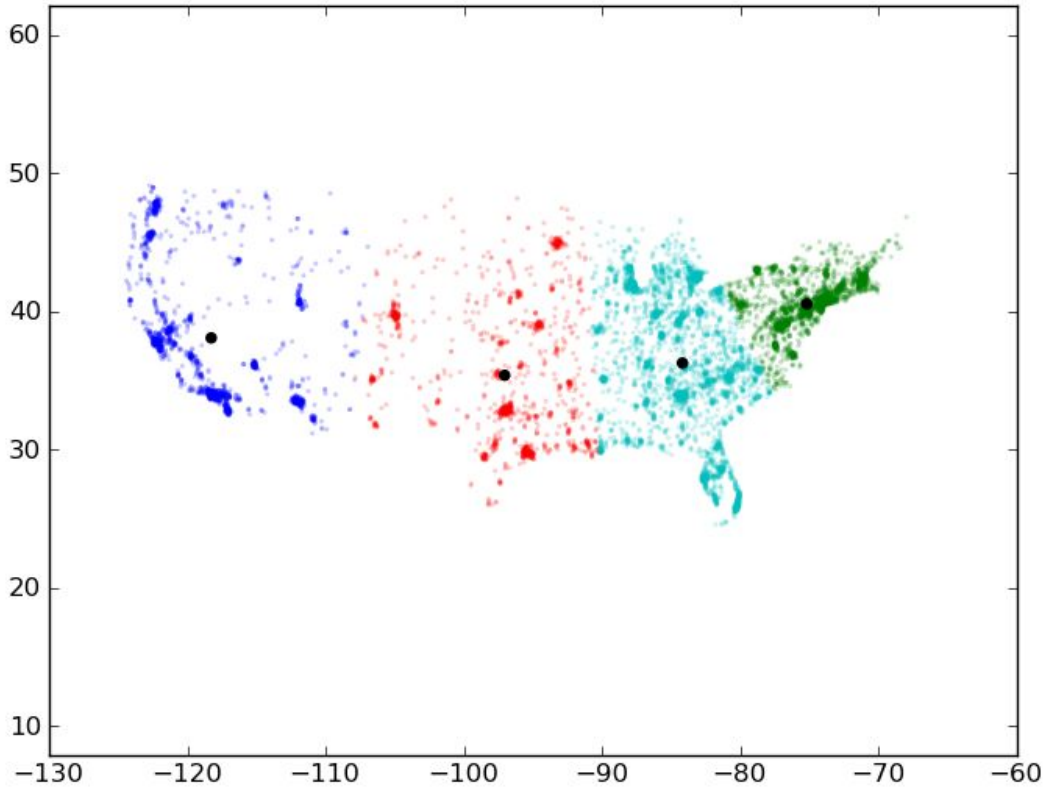
K = 2 GreatCircle:



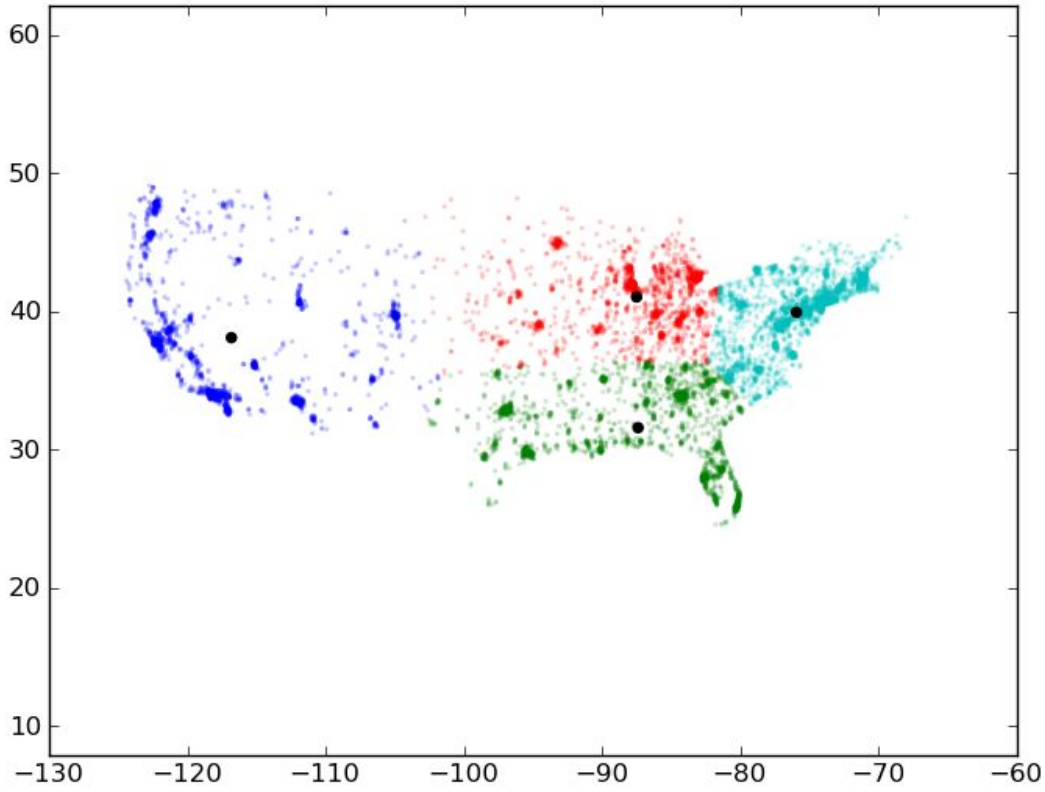
Results/discussion

When K = 2, there are no observable differences between Euclidean measure and Great Circle measure. Both of them split the US into west part and east part.

K = 4 Euclidean:

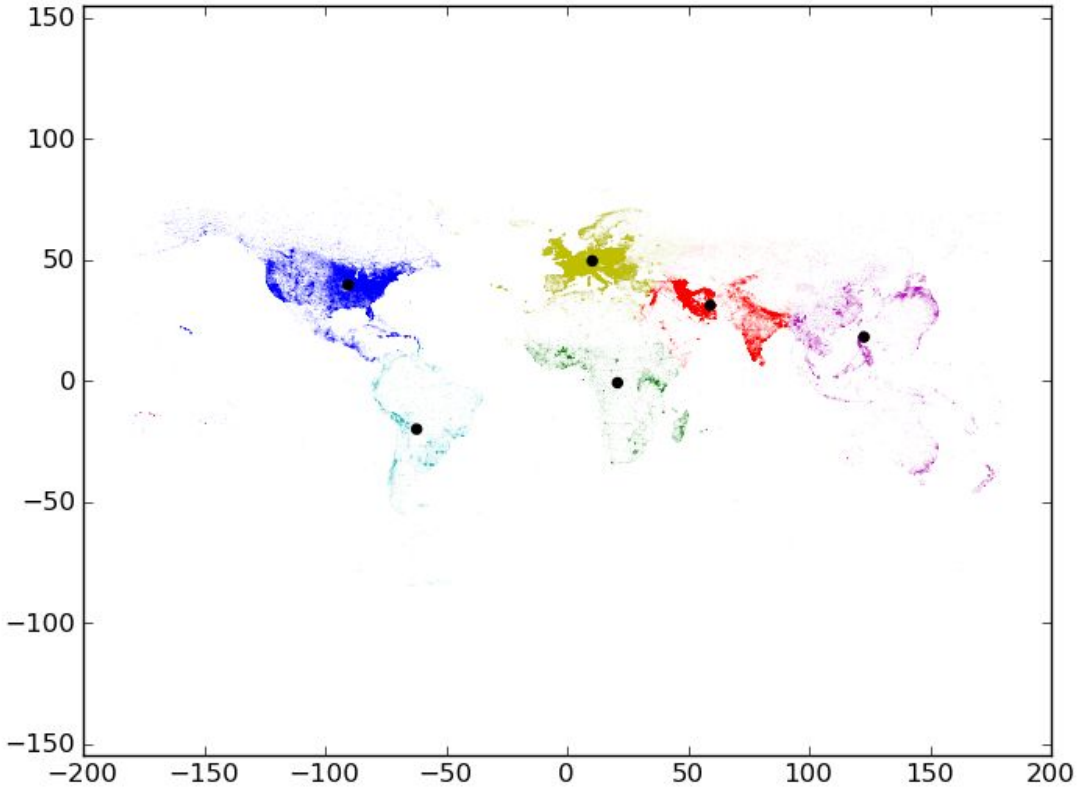


K = 4 GreatCircle:

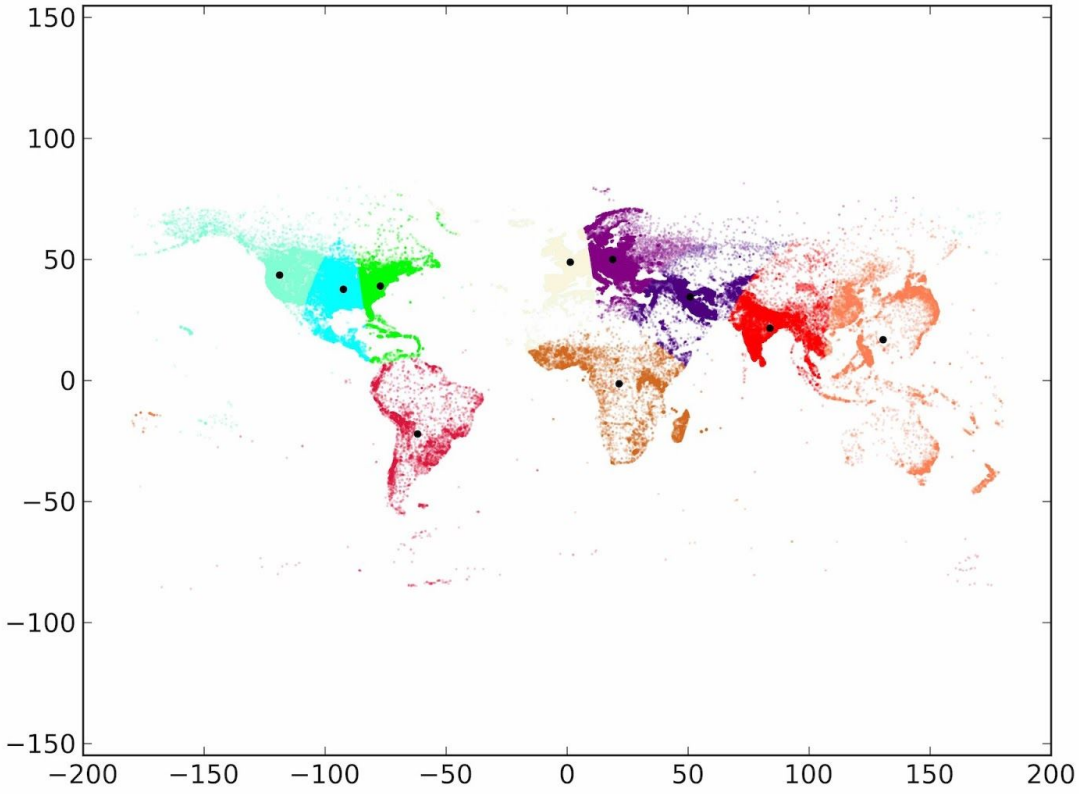


results/discussion
When K = 4, the differences seem clear. For Euclidean measure, the US is split into 4 parts mainly in a longitudinal way. The US is split into the west, the central west, the central east, the east. For Great Circle measure, the US is split into 4 parts in both longitudinal and latitudinal way.

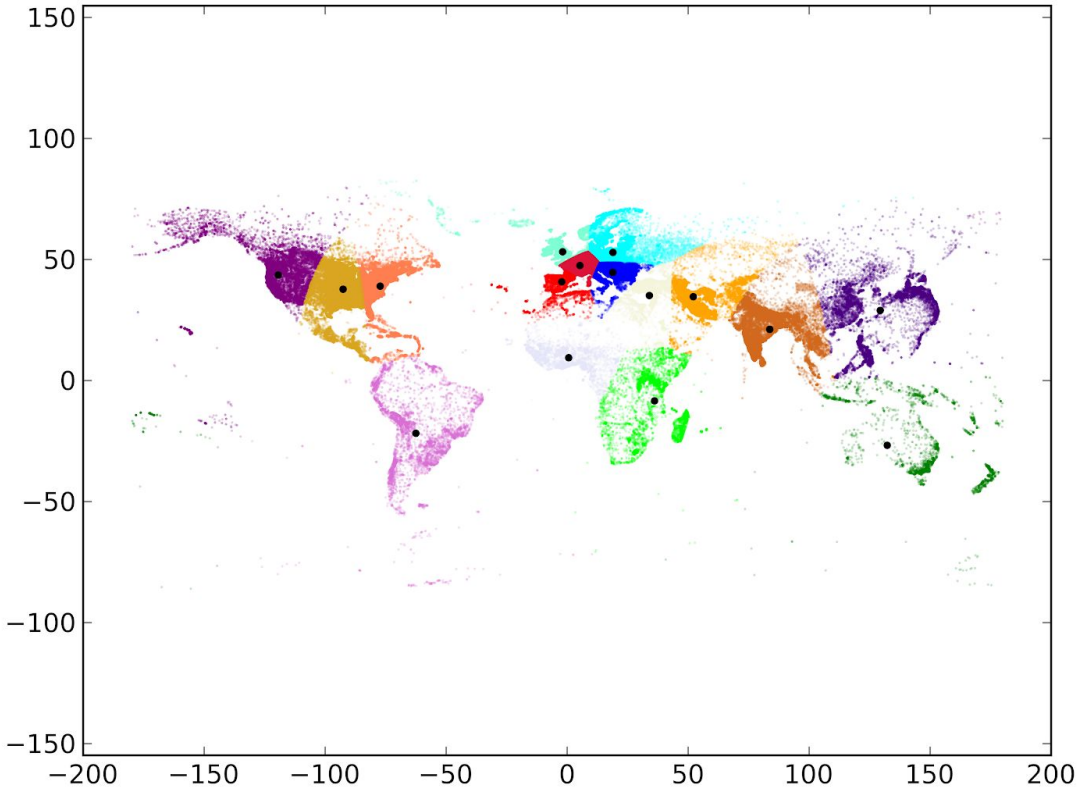
Dbpedia
K = 6 GreatCircle



K = 8 GreatCircle



K = 16 GreatCircle



When k = 6, the world is split into 6 parts and each cluster represent North America, South America, Europe, Middle East, Africa and Asia.

For k = 8, all the clusters remain the same as k = 6 except North America. North America is split into 3 parts in a more detailed way. It seems that there are lots of data set in North America and this region requires further clustering.

For k = 16, the world is split even more specifically. Almost every region contains several parts, especially Europe.

I think k = 6 makes sense considering it effectively split each continent. Since the data represents a location that has a wikipedia page. Therefore, when k increases, the region that contains more locations edited on wiki page will be more likely to become a cluster.

Runtime Analysis

	devicestatus		Synthetic		dbpedia
k	5	5	5	5	5
measure	EU	GC	EU	GC	GC
Time (no cache)	3.0min	9.4min	2.9min	3.5m	6.5min
Time (with cache)	2.4min	4.0min	39s	1.1m	4.2min

* EU is Euclidean and GC is GreatCircle
* randomly pick centroids first, then fix them for one dataset for the sake of testing(ie):
Clocations = [(-90.171112059999999, 30.092735650000002), (-75.362937930000001, 39.725557299999998), (-117.9957428, 34.056018379999998), (-112.404335, 33.577716909999999), (-87.176986690000007, 37.014423409999999)]
*settings(eg computers) for each column may not be the same. It is allowable since we are interested in comparing computation time with persist() and without it.

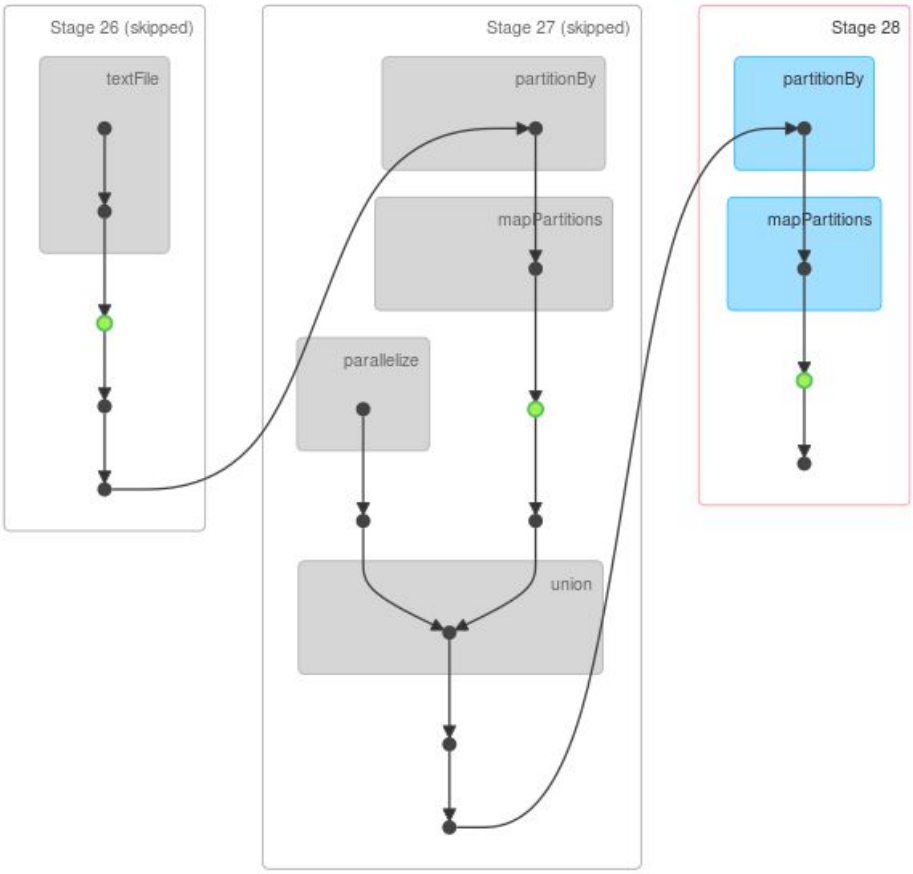
Discussion

Caching:

-caching is very import in reducing run time
-persist() is a way to store data locally in the node, it will chek if the size will fit in memory. If so, data will be store in memory. If not, it will be stored in disk. cache() is similar, specificlly just the first case.
-It is important to consider which rdd to persist() and where it is stored under the hood as well. For example, If the data A is not too big we can persist the data A. If data A is too big, we can persist alternative data B if possible. In our case, data A is data containing all the points and data B is information of centroids.
-Here we have a small K, so centroid's information can be easily cached in memory.

Lessons:

-The job info is available in http://quickstart.cloudera:18088/ . It shows the stages and corresponding time and code, as the figure shows below:
-Avoid unnecessary read/write if possible. It shows that in the step of counting total numbers for calculating new centroid location takes a lot of time.



-

Big data application/dataset

Gowalla Location-based online social network dataset

We use a dataset in the website below:
https://snap.stanford.edu/data/loc-Gowalla.html

If we have a geo-location dataset of an App user, we can try to inteprete the clustering result. First of all we can at least understand some baisc demographics of the App user. Furthurmore, we can have research on maketing, user-experience and so on.

Description

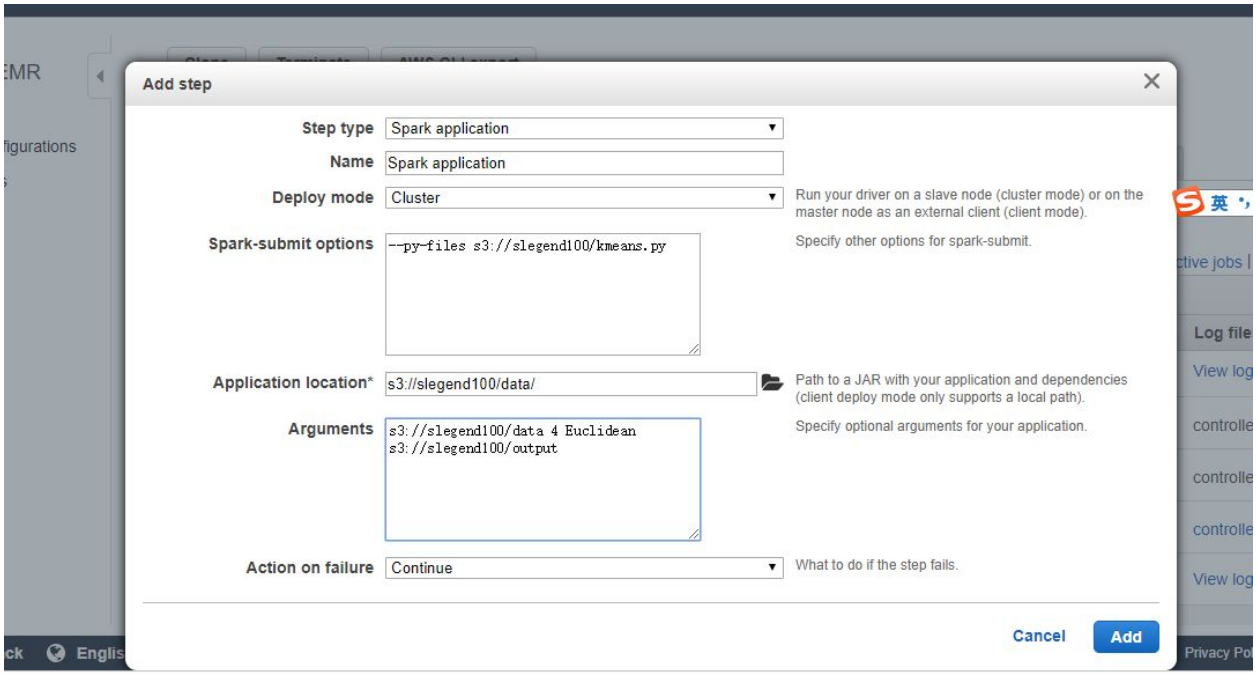
Gowalla is a location-based social networking website where users share their locations by checking-in. The friendship network is undirected and was collected using their public API, and consists of 196,591 nodes and 950,327 edges. We have collected a total of 6,442,890 check-ins of these users over the period of Feb. 2009 - Oct. 2010.

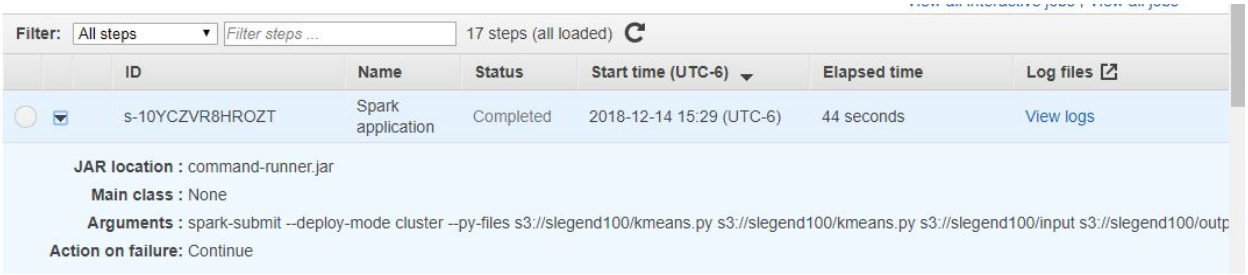
Dataset statistics	
Nodes	196591
Edges	950327
Nodes in largest WCC	196591 (1.000)
Edges in largest WCC	950327 (1.000)
Nodes in largest SCC	196591 (1.000)
Edges in largest SCC	950327 (1.000)
Average clustering coefficient	0.2367
Number of triangles	2273138
Fraction of closed triangles	0.007952
Diameter (longest shortest path)	14
90-percentile effective diameter	5.7
Check-ins	6,442,890

Example of check-in information

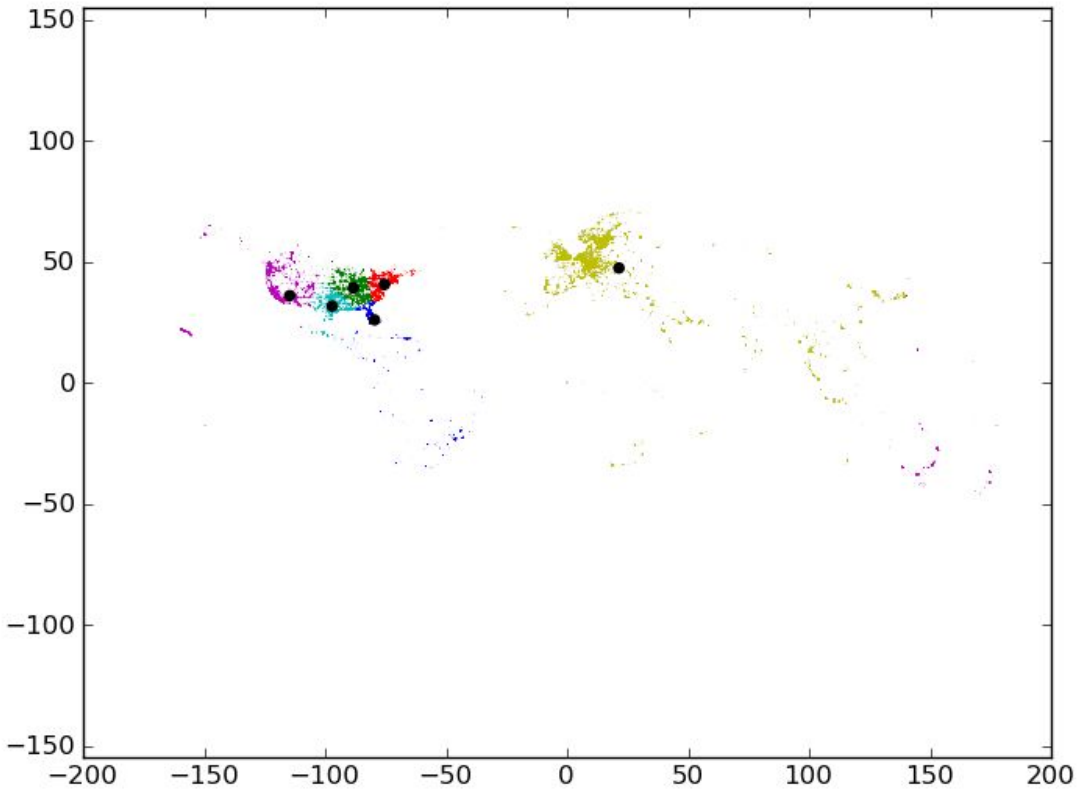
[user]	[check-in time]	[latitude]	[longitude]	[location id]
196514	2010-07-24T13:45:06Z	53.3648119	-2.2723465833	145064
196514	2010-07-24T13:44:58Z	53.360511233	-2.276369017	1275991
196514	2010-07-24T13:44:46Z	53.3653895945	-2.2754087046	376497
196514	2010-07-24T13:44:38Z	53.3663709833	-2.2700764333	98503
196514	2010-07-24T13:44:26Z	53.3674087524	-2.2783813477	1043431
196514	2010-07-24T13:44:08Z	53.3675663377	-2.278631763	881734
196514	2010-07-24T13:43:18Z	53.3679640626	-2.2792943689	207763
196514	2010-07-24T13:41:10Z	53.364905	-2.270824	1042822

Inplementation

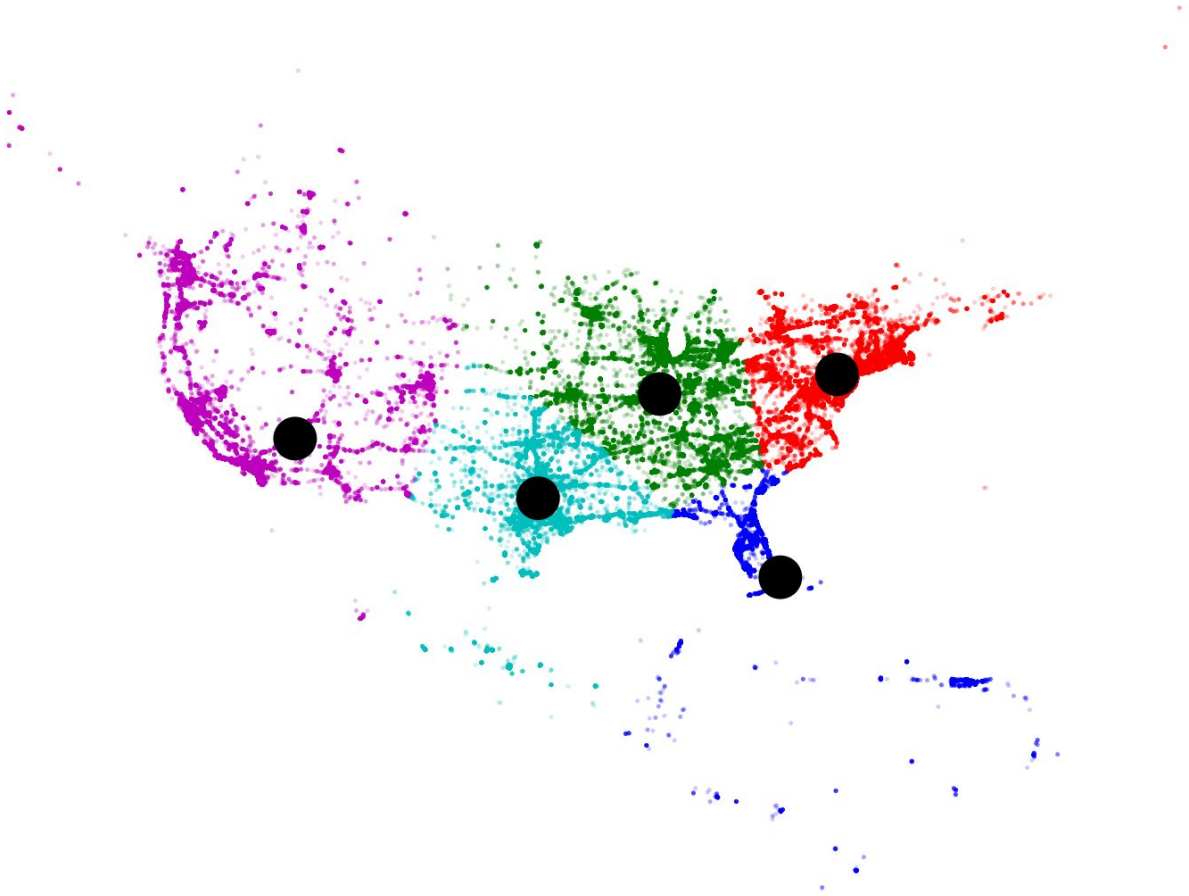




Result and discussion



-From this figure we can see that the majority of the user check in US and Europe.



-From a zoom-in figure, we can observe some pattern of routes. They may be some popular routes for travelling. Since it might be the case that users of the App typically check in while they are travelling.

-If we even zoom in more or confine our area, we can probably learn which places are densely filled. And it may indicate the place is more social friendly /polular /there are many tourists or young people since this is a social App.

Conclusion

In our project, we successfully scales the implementation from local to the cloud. Becasue the size of the data is bigger and bigger, cloud computing becomes more popular way to implement.

And we follow the procedure of test locally -> test in pseudo-distributed -> test on cluster -> run on cluster. The most important parts in the procedure were testing locally and testing on the cluster. Testing locally can reduce the cost of determining k running in the cluster. Testing cluster helps us to ensure that we had the correct setup on the cluster before running the full job. However it is definately helpful if we were able to evenly take sample for big data.

Besides, we find that cloud computing is a really useful tool for data of larger scale. With data of Mbs we can directly work in python. But for big data or industry-level data, we definately need more powerful tools. The data is bigger and bigger these days, using cloud computing is really the fashion now and in the future.

Lessons learned

-Caching in cloud computing is important
-It is up for us to decide the number of clusters that makes sense. Given a context or a client, hopefully there is a good idea of K. For example, if we want to cluster for T-shirt size, we can set K as 4 because maybe we want sizes of “S, M, L, LX”. To conclude, we need domain knowledge or an intent if we want to decide K easily.
-May use Hierarchical Clustering to produce dendrogram to help decide K if possible.
-Avoid unnecessary rdd actions as much as possible.

Future work

-Clustering with data of higher dimensions.
-We can plot the data in the sphere rather than 2-D plot.
-Reduce the amout of calculation to improve the runtime.
-Besides that, we can use other clustering algorithm such as EM, kmedian clustering, Mean-Shift Clustering, Density-Based Spatial Clustering of Applications with Noise (DBSCAN) etc.