Evaluating Clustering Techniques over Big Data in Distributed Infrastructures

by

Kartik Shetty

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

____

April 2018

APPROVED:

____

Professor Mohamed Y. Eltabakh, Major Thesis Advisor

____

Professor Dmitry Korkin, Thesis Reader

____

Professor Craig Wills, Head of Department

Abstract

Clustering is defined as the process of grouping a set of objects in a way that objects in the same group are similar in some sense to each other than to those in other groups. It is used in many fields including machine learning, image recognition, pattern recognition and knowledge discovery. In this era of Big Data, we could leverage the computing power of distributed environment to achieve it over large dataset. It can be achieved through various algorithms, but in general they have high time complexities. We see that for large datasets the scalability and the parameters of the environment in which it is running become issues which needs to be addressed. Therefore it's brute force implementation is not scalable over large datasets even in a distributed environment, which calls the need for an approximation technique or optimization to make it scalable.

We study three clustering techniques: CURE, DBSCAN and k-means over distributed environment like Hadoop. For each of these algorithms we understand their performance trade offs and bottlenecks and then propose enhancements or optimizations or an approximation technique to make it scalable in Hadoop. Finally we evaluate it's performance and suitability to datasets of different sizes and distributions.

Acknowledgements

I am grateful to my advisor Prof. Mohamed Y. Eltabakh, for his guidance and words of encouragement during my studies. I am indebted to him for introducing me to the field of Big Data without which this thesis would not have been possible. His ability to convey ideas and patiently direct my work has helped me immensely to define the scope of my work for which I offer my sincerest appreciation.

I appreciate my friends, Shreyas and Sumedh who have encouraged me to continue whilst hitting roadblocks. Thanks to Mahesh, Mukesh, Kenil, Saran and Divya for your friendship.

I would like to thank my uncle and my aunt for supporting and encouraging me to pursue my graduate studies.

Finally, I would like to thank my mother and sister for being my pillars of support and for making sacrifices without which I would not have been able to pursue my dreams.

Contents

1 Introduction **1**

1.1 CURE 3

1.2 DBSCAN 4

1.3 k-means 5

1.4 Challenges 5

2 Background **8**

2.1 CURE 8

2.1.1 Sampling 8

2.1.2 Partitioning 9

2.1.3 CURE Algorithm 9

2.1.4 Merge Process 11

2.1.5 Outliers Elimination 13

2.1.6 Data Structures 13

2.2 DBSCAN 15

2.3 k-means 17

2.3.1 Algorithm 17

2.3.2 Choosing K 19

3 Hadoop and MapReduce paradigm **20**
3.1 Apache Hadoop 20
3.1.1 High Level Architecture of Hadoop 21
3.1.2 HDFS 22
3.1.3 MapReduce paradigm 24
4 Map Reduce Implementations **26**
4.1 CURE 26
4.2 Bottlenecks and Challenges 27
4.2.1 k-merge Implementation 28
4.2.2 Hash Map Implementation 28
4.3 DBSCAN 32
4.3.1 Data Partitioning 32
4.3.2 DBSCAN Clustering 35
4.3.3 Merge Phase 36
4.3.4 Relabel Data 37
4.3.5 Optimization 38
4.3.6 Objective 39
4.4 kmeans 40
4.4.1 Optimization 41
5 Experiments and Evaluation **44**
5.1 CURE 44
5.1.1 Experimental Setup 44
5.1.2 Datasets 44
5.1.3 Input parameters 45
5.1.4 Results and Evaluation 45

5.1.5 Custom Dataset 46
5.1.6 Real World Dataset 48
5.2 DBSCAN 49
5.2.1 Experimental Setup 49
5.2.2 Dataset 49
5.2.3 Results and evaluation 50
5.2.4 Evaluation 51
5.3 k-means 51
5.3.1 Set up 52
5.3.2 Dataset 52
5.3.3 Results 52
5.3.4 Evaluation 53

6 Related Work **54**
6.1 CURE 54
6.2 K-Means 54

7 Conclusion and Future work **55**
7.1 CURE 55
7.2 DBSCAN 56
7.3 k-means 57

A Basic sequential algorithmic scheme **58**

List of Figures

1.1 Stages in CURE . . . 3
1.2 Classification of points in DBSCAN . . . 4
1.3 Steps in k-means . . . 6

2.1 flowchart CURE . . . 11
2.2 Representation of points (2,3),(4,7),(5,4),(7,2),(9,6),(8,1) in k-d tree . 14
2.3 Reachability . . . 16

3.1 Hadoop Architecture . . . 21
3.2 HDFS Architecture . . . 23
3.3 Stages in MapReduce . . . 25

4.1 k-merge implementation of CURE . . . 29
4.2 hashMap implementation of CURE . . . 31
4.3 Stages in DBSCAN . . . 32
4.4 Steps in Data Partitioning . . . 33
4.5 Partitioning overview . . . 35
4.6 Mapper in kmeans . . . 40
4.7 Reducer in kmeans . . . 41
4.8 Use of Combiner . . . 42

5.1 Results for K- Means . . . 53

List of Tables

5.1 Sample rate:1% 46

5.2 Sample rate:5% 46

5.3 Sample rate:10% 46

5.4 Sample rate:20% 47

5.5 Sample rate:60% 47

5.6 Silhouette coefficient calculated for sequential clustering 47

5.7 Real World Dataset Results 48

5.8 *epsilon*:0.1 50

5.9 *epsilon*:0.01 50

5.10 *epsilon*:0.001 50

5.11 *epsilon*:0.0001 50

5.12 Using the combiner 52

5.13 Without using the combiner 53

Chapter 1

Introduction

Clustering is one type of unsupervised learning methods that is used for discovering useful data distribution and patterns in the underlying data. Given a set of n points, the task of clustering can be described as to partitioning points into k partitions or groups based on their similarity. Additionally some of the clustering methods take k as an user defined parameter while others detect the number of partitions automatically. For example: k-means[7] takes k as an user defined input while DBSCAN[5] achieves it automatically. Based on the methodology used in the technique, the clustering algorithm can be classified into Hierarchical based, Density based, Centroid-based and Distribution based.

We are concerned with the first three methods of the above mentioned one's. Hierarchical based clustering performs hierarchical decomposition of given data points. Data points having similar features are merged at different levels forming tree shaped structure. Initially each data point is considered as a self made cluster and then iteratively clusters are merged based on common feature or pattern to form a tree like structure. In hierarchical clustering there are different algorithms that have been proposed like CLARAN [11], CURE [6], CHAMELEON[9] and BIRCH [13].

Hierarchal algorithms are not scalable to apply on large datasets since it has a time complexity of at least O(n^2).

In density-based clustering, we identify areas with high density of data to clusters and the sparse ones are identified as noise or outliers. The examples of this method include DBSCAN [5], OPTICS [2] and Mean-shift algorithm [4]. In centroid-based clustering, we represent clusters using central vectors which may not be a member of the dataset. We assign data to clusters based on the minimum distance to these central vectors. k-means algorithm [7] comes under kind of clustering technique.

Distributed computing environment which consists of interconnected nodes can be used to apply clustering algorithms to large amount of data. Typically in this environment we partition the dataset and then process each partition in parallel using the nodes and then combine the results from each node. The idea here is to leverage the parallel computing power in hand, but since the algorithms have time complexities we need to propose an enhancement or an optimization to make it scalable. Apache Hadoop, is an open source framework for distributed computing which uses Map Reduce paradigm and we will be using it for implementation.

Our work involves implementing the three clustering algorithms: CURE, k-means and DBSCAN in a distributed environment like Hadoop. For each of these algorithms we understand their performance trade offs and bottlenecks and then propose enhancements or optimizations or an approximation technique to make it scalable in Hadoop. Finally we evaluate it's performance and suitability to datasets of different size and distributions.

In the following subsections we discuss the three techniques in an overview and the challenges one faces while implementing them in a distributed environment.

1.1 CURE

CURE is a hierarchical algorithm which utilizes representative points which are selected by obtaining the most scattered points within a cluster and then shrinking them towards the mean of the cluster by a specified fraction. This enables to determine clusters of non-spherical sizes and the ones having wide variances in size[6]. The stages involved in the algorithm are depicted in the Figure 1.1.

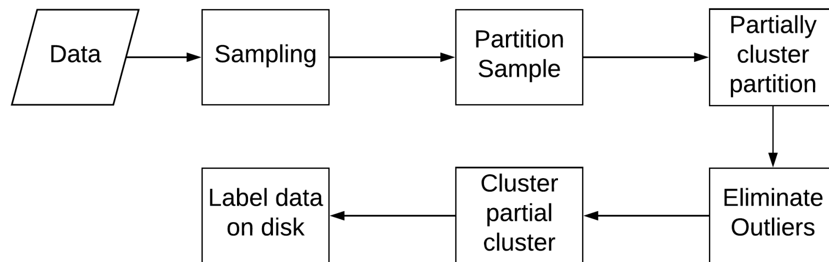


Figure 1.1: Stages in CURE

Firstly, the data is sampled using a sampling technique. Secondly the sampled data is partitioned to k partitions. Thirdly, each partition is processed using the CURE algorithm until it is paused to check for outliers, after the removal of outliers the clustering is resumed until we get the required number of clusters. The clusters from each partitions are combined using the CURE algorithm to get the final list of clusters. Finally the data points on the disk which were not selected for sampling are assigned clusters from the above step. Since it uses a combination of partitioning and sampling it is a good candidate among the hierarchical algorithms to be implemented in a distributed environment.

CURE uses a k-d tree and a priority queue to process the clusters. The k-d tree is used to retrieve the closest cluster for a given cluster, the search operation has a time complexity of O($logn$). The priority queue stores the list of clusters ordered with respect to the distance of a given cluster to its closest cluster. The algorithm runs

till the number of clusters in the queue equals to the number of clusters required. The algorithm has a time complexity of O($n^2 logn$) and a space complexity of O(n).

1.2 DBSCAN

DBSCAN[5] is a density-based clustering technique, which takes in minPnts and ϵ as input parameters. Using this technique for every data point considered, we would draw an imaginary circle with radius ϵ and then perform a neighborhood query to retrieve the number of points within the imaginary circle drawn. Based on the number of points returned we would classify the point as a core point, border point or a noise point. For example in the figure 1.2 for inputs *minPnts*=6 and ϵ radius,

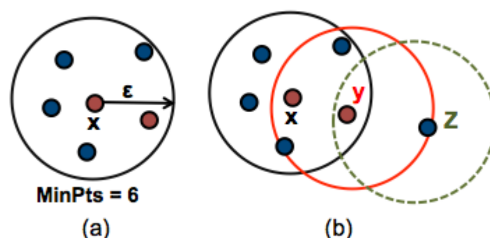


Figure 1.2: Classification of points in DBSCAN

we classify x as a core point as *neighborhood(x)* >= *minPnts*, y is classified as a border point as it is in the neighborhood of the core point x and *neighborhood(y)* < *minPnts* and z is classified as a noise point as *neighborhood(z)* < *minPnts* and it not in the neighborhood of a core point.

The points returned from the query are inserted into a queue, the same procedure is repeated for each point in the queue in a iterative manner until the queue is empty and all the points processed are assigned to a cluster. The procedure resumes with the next unprocessed point in the dataset until all the points are processed. This approach has a time complexity of O($nlogn$) if proper index structure is used and a meaningful value of ϵ is chosen.

There are three main advantages of using this approach. Firstly it is insensitive to the ordering of points, as it does not matter the order in which we process the points as the output is going to be the same. Secondly we need not specify the number of clusters prior to clustering. Thirdly, it is capable of identifying clusters of arbitrary shape. The disadvantage of this approach is that it is very sensitive to the parameter settings, for example: for two sets of values of minPnts and ϵ we are going to get different set of clusters.This makes the approach nondeterministic.

1.3 k-means

k-means[7] is a centroid based clustering technique, it takes seeds as input which are central vectors and the final number of clusters produced is equal to the number of seeds provided. Every point in the dataset is assigned to one of the seeds by determining the one with the minimum distance to it. Once all the points are assigned clusters, the central vectors are calculated again by determining the centroids of the assigned clusters. This process is done in an iterative manner until the newly calculated central vectors are equal to the previous one or if the number of iterations is equal to a bound which is provided as an input. The steps involved in the algorithm are depicted in the figure

1.4 Challenges

The are three obstacles one faces while implementing a clustering algorithm over a distributed environment. Firstly, most of the clustering algorithms have a high time complexity hence a brute force implementation is not going to be scalable for large input sizes. This calls the need for an optimization or an approximation

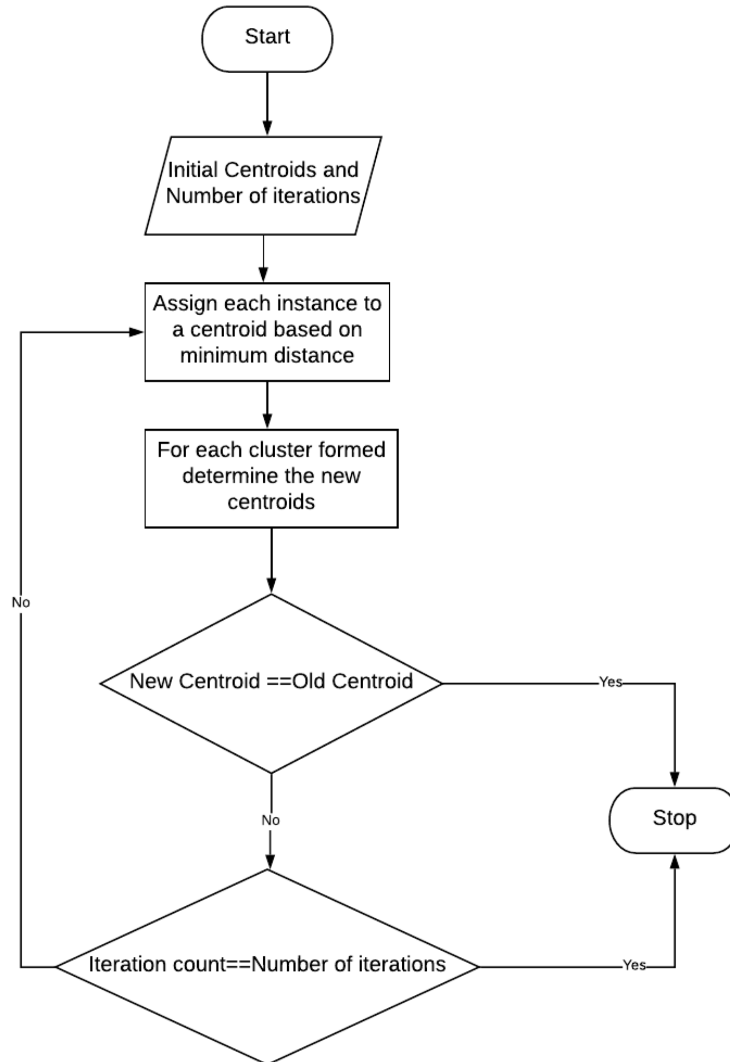


Figure 1.3: Steps in k-means

technique to be applied so that the clustering becomes scalable over a distributed environment. Secondly, there is a need for the different stages involved in a clustering algorithm to be interpreted in terms of Map Reduce paradigm since a distributed environment like Hadoop uses this paradigm. For example : sampling is one of the stages involved in the CURE clustering technique, we would have to decide how this could be accomplished in a Map Reduce paradigm.

Thirdly, even after coming up with the distributed version of the algorithm we

find that the environment parameters and nature of the dataset has an effect on the performance of the algorithm on a distributed environment.

Chapter 2

Background

2.1 CURE

CURE as described in the previous section is a hierarchical clustering algorithm. The Figure 1.1 shows the various stages involved and we are going to explore each stage in detail.

2.1.1 Sampling

In this phase we use a technique to extract a sample of the whole data input. Hence this phase is called "Sampling". We use a technique called "Reservoir Sampling" to accomplish the task of selecting samples. Reservoir Sampling comes under a family of randomized algorithms to select k sample from a list of n items, where n is large and is not small enough to fit into main memory. The steps involved in the algorithm are listed in Algorithm 1. The algorithm begins by initializing the output with the first k elements with an index i which iterates from 0 to k-1. We proceed by iterating index i from k-1 to n-1, where n is the total size of the input. For every iteration we determine a random number r between 0 to i and if it happens that

r is less than k, we assign the ith element in the input to the rth element in the output. The time complexity of this approach is O(n) and the space complexity of this approach is O(k).

Algorithm 1: Reservoir Sampling

Result: reservoir
Input : input[]:Array containing the input,k:number of partitions,n:number of elements in the input array
Output: Array of sampled input of size k

```
1 i ← 0 ;
2 reservoir[] ← Array of size k;
3 for i =0 to k-1 do
4     reservoir[i] ← input[i] ;
5 RandomNumberGenerator r ;
6 for i = k to n-1 do
7     j ←r.nextInt(i+1);
8     if j¡k then
9         reservoir[j] ← input[i] ;
```

2.1.2 Partitioning

This phase assigns the points inside the input to its appropriate partitions. The intention to create partitions is so that we can process the partitions independently and in a parallel fashion. We use straight forward approach, which process's the data points in sequential and assigns partition using the equation.

$$Partition\,Number \leftarrow Input\,index \bmod Partition\,Size \tag{2.1}$$

2.1.3 CURE Algorithm

CURE is a hierarchical algorithm which utilizes representative points which are selected by obtaining the most scattered points within a cluster and then shrinking

them towards the mean of the cluster by a specified fraction. This enables to determine clusters of non-spherical sizes and the ones having wide variances in size[6]. The algorithm presumes each point to be a cluster of its own, and then proceeds by inserting these clusters into the k-d tree T and the priority queue Q. It inserts a cluster c into the queue Q based on the distance to its closest cluster. It removes the cluster u from the priority queue and determines the its closest cluster v. The representative points from the clusters u and v are removed from the k-d tree. The merged cluster w is formed by selecting c closest representative points from the set of representative points from u and v. It proceeds by iterating through the queue, and for each cluster x we update its closest cluster as there are two scenario's now:first one is that the merged cluster w might be its closest cluster and second one is that it might have u or v as its closest cluster and since those clusters are merged there is a need to update. For the second scenario, according to [6] there is bounding distance which is defined as the distance between merged cluster w and cluster x under consideration, this optimizes the search of the closest cluster by avoiding the need to iterate through all the nodes of the k-d tree. This process is repeated until we get the required number of clusters. So for every merged cluster w produced we iterate through all the elements in the priority queue and in the worst case we might use the k-d tree for the search operation which gives a time complexity of O($nlogn$). Therefore for n merged clusters we get a time complexity of O(n^2logn), which is quite high. The flowchart in Figure 2.1 presents a high level view of the steps involved in this implementation.

The pseudo code of the approach is presented in algorithm 2.

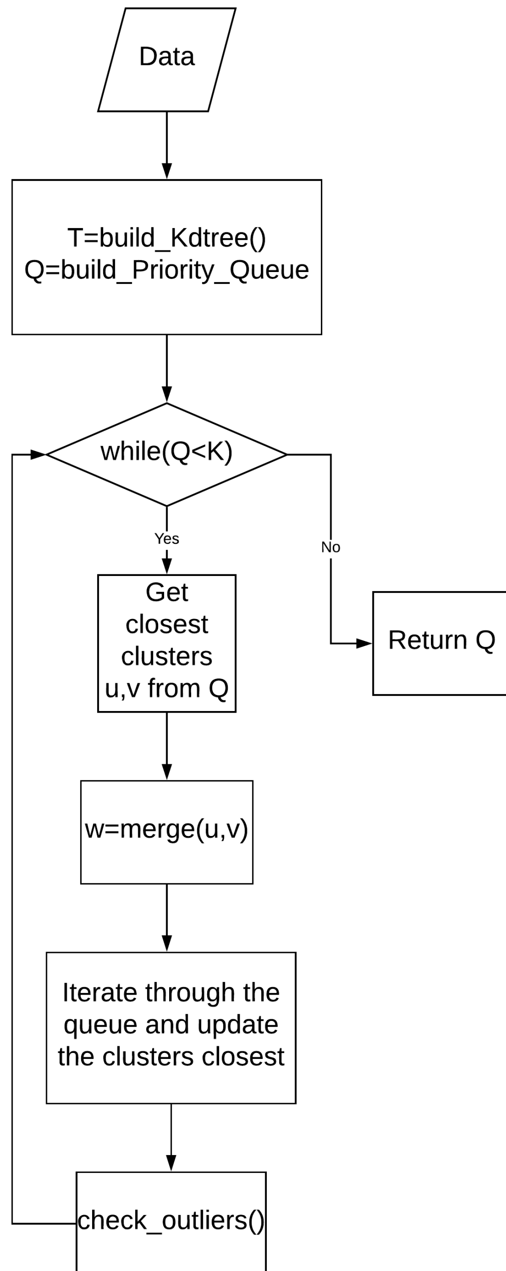


Figure 2.1: flowchart CURE

2.1.4 Merge Process

We use the improved merge procedure as discussed in [6] while merging two clusters. We essentially do not store the points that are assigned to the clusters, instead we

Algorithm 2: CURE

Result: Q
Input : S:list of points,k:number of clusters,alpha:shrinking factor,c:number of
Output: Set of clusters of size k

```
1  T ← buildkd_tree(S);
2  Q ← buildheap(S);
3  while Q.size() > k do
4      u ← Q.poll();
5      v ← u.getClosest();
6      Q.remove(v)
7      Delete from priority queue;
8      T.delNode(u.getRep());
9      T.delNode(v.getRep());
10     w ← merge(u, v, c, alpha);
11     T.insertNode(w.getRep());
12     w.setClosest(Q.peek());
13     foreach x in Q do
14         if dist(w, x) < dist(w, w.closest) then
15             w.closest ← x;
16             if x.closest is either u or v then
17                 x.closest ← closest_cluster(T, x, dist(x, w));
18             else
19                 x.closest ← w;
20             relocate(Q, x);
21         else
22             if dist(x, x.closest) > dist(x, w) then
23                 x.closest ← w;
24                 relocate(Q, x);
25     insert(Q, w);
```

only store the representative points. Suppose we have clusters u and v merged, as each cluster has utmost c representative points, there is a need to iterate though only 2c points instead of n points, there by the complexity comes down to O(1) instead of O(n). We calculate the scattered point by unshrinking the representative

points using the (2.2).

$$p = p + alpha * (w.mean - p) \tag{2.2}$$

where w.mean is the mean of the merged cluster and alpha is the shrinking factor. The algorithm 3 describes the process.

Algorithm 3: Merge Procedure

Result: w
Input : the clusters to be merged u and v,alpha
Output: Merged cluster w

1. $w \leftarrow u \cup v$;
2. $w.mean$ ←calculate the mean;
3. *tempset* ←determine the c closest points among 2c points from u and v;
4. **foreach** *p in tempset* **do**
5. $\quad w.rep \leftarrow w.rep \cup (p + alpha * (w.mean - p))$;
6. *return* w;

2.1.5 Outliers Elimination

When the number of clusters formed is equal to the 1/3 of the initial number of clusters we started with, in this case it would be the number of points we began with. We would remove the clusters having less than 2 representative points. Once the outliers are removed, we resume with the clustering of the points using CURE.

2.1.6 Data Structures

k-d tree

This data structure is used to index the points and to optimize the nearest neighborhood query. The k-d tree is a binary tree in which every node is a k-dimensional point. Every non-leaf node can be thought of as generating a splitting hyperplane

that divides the space into two parts, Points to the left of this hyperplane are represented by the left subtree of that node and points right of the hyperplane are represented by the right subtree. So, for example, if for a particular split the "y" axis is chosen, all points in the subtree with a smaller "y" value than the node will appear in the left subtree and all points with larger "y" value will be in the right subtree. The following are the time complexities for the basic operations:

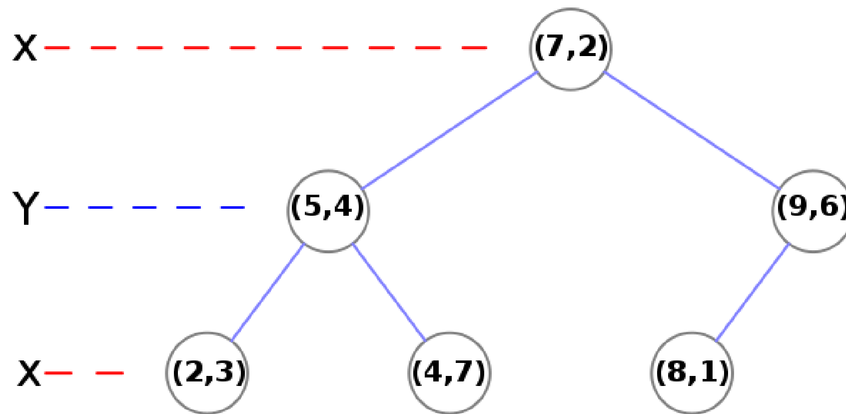


Figure 2.2: Representation of points (2,3),(4,7),(5,4),(7,2),(9,6),(8,1) in k-d tree

- Insertion:Insertion of n points will take O(nlog^2n)) if O($nlogn$) sort is used to determine the median at every level.
- Deletion: Removal of a point in the tree will take O($logn$) where n is the number of points in the tree.
- Insertion of single point:O($logn$)
- Nearest neighbor search:O($logn$)

Priority Queue

Priority Queue is min-heap or max-heap data structure and is used in the algorithm to store the clusters in the order of its distance to its closest cluster. The following

are the time complexities for the operations on it:

- fin-min:O(1)
- delete-min:O($logn$)
- insert:O($logn$)
- decrease key:O($logn$)

2.2 DBSCAN

DBSCAN is a density based clustering technique[5]. It is comes under density based clustering technique because for every point it process's it draws an imaginary circle with ϵ radius and counts the points under that region. Based on the number of points it categorizes the points to following three types:

- Core point: x is a core point if *neighborhood(x)* $>=$ *minPnts*
- Border point:x is a border point if *neighborhood(x)* $<$ *minPnts* but is in the neighborhood of a core point.
- Noise point:x is a noise point if *neighborhood(x)* $<$ *minPnts* and is not in the neighborhood of a core point.

Before processing, for all the points in the dataset we set the status as unprocessed. We take a random point from the dataset and change its status to processed. We then query the points in its neighborhood within ϵ distance by drawing an imaginary circle of ϵ. The points returned from the query are inserted into a queue, the same procedure is repeated for each point in the queue in a iterative manner until the queue is empty and all the points processed are assigned to a cluster. And then we

proceed to the next unprocessed point. The concept of reachability explains how we assign points to a cluster. The following are the three types of reachability and is also depicted in the Figure2.3:

- Directly reachable: if for a point p, point q falls within the ϵ radius, it is said that q is directly reachable from p.

- Density reachable: if for a point p, point q does not fall within the ϵ radius, but it is directly reachable from point o which is directly reachable from point p. Point p and q are said to density reachable.

- Density connected: if point p and point q are density reachable from point o, then p and q are said to be density connected.

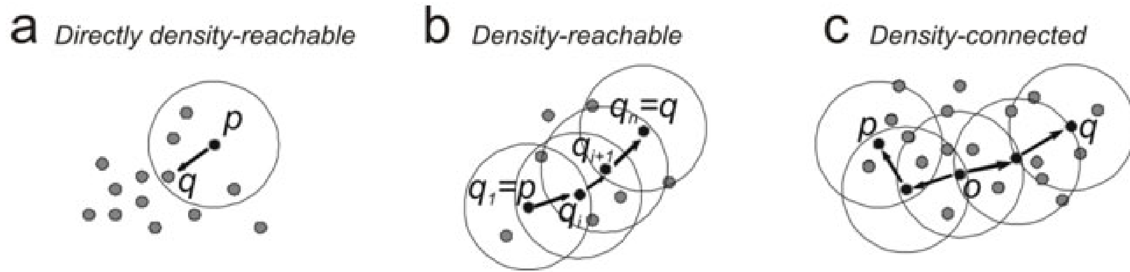


Figure 2.3: Reachability

A cluster C with respect to ϵ and *minPnts* in D, where D is the set of objects satisfy the following conditions[5]:

- Maximality:$\forall$ p,q $\in$ D:if p$\in$C and q is density reachable from p with respect to ϵ and *minPnts*, then q $\in$ C.

- Connectivity:$\forall$ p,q $\in$ C: p is density connected to q with respect to ϵ and *minPnts*. Every object not contained in any Cluster is noise.

The algorithm 4 describes each step in clustering. It begins by building a R-tree of the points. It is a spatial index data structure which can perform neighborhood

search query in O(log n) time. We can observe that for a point p which is unvisited it performs a neighborhood query and if the number of points returned is greater than minPnts it assigns the same clusterID for its neighbors which ensures the two properties we discussed above: Maximality and Connectivity. Also we can observe that the algorithm gives the same result and is not dependent on the order of the points. For example: if a point p is assigned a type NOISE but is in fact a BORDER point, line number 21 in algorithm 4 ensures that it is assigned the right type. The time complexity of the approach is *(nlogn)*, as for every unprocessed point we perform the neighborhood query which has time complexity of *(logn)*.

2.3 k-means

k-means is a type of unsupervised learning, which is used to group data where the number of groups is represented by the variable **K**. The algorithm works iteratively to assign every data point to one of the group based on feature similarity. The results of the algorithm are :

- Centroids of K clusters which can be used to label data.
- Data points assigned to one of the K clusters.

Also examining the results of the centroid values can be used to interpret the kind of group each cluster represents.

2.3.1 Algorithm

The algorithm runs in a iterative manner in such a way that the results after every iteration converges towards the final result. The input to the algorithm are the

Algorithm 4: DBSCAN

```
   Result: DB
   Input  : DB:p1,p2..pn, ϵ and minPnts
   Output: each point p assigned to type(BORDER,CORE,NOISE) and a
           clusterID
 1 rtree ←buildRtree(DB);
 2 clusterID ← 0;
 3 foreach unvisited point p in DB do
 4     mark p as visited;
 5     nbhdP ←getNeighborhood(p,ϵ);
 6     if SizeOf(nbhdP) < minPnts then
 7         p.flag ←NOISE;
 8     else
 9         p.flag ←CORE;
10         p.clusterID ←clusterID;
11         foreach point q in nbhdP do
12             if q is unvisited then
13                 mark q as visited;
14                 q.clusterID ←clusterID;
15                 nbhdQ ←getNeighborhood(q,ϵ);
16                 if SizeOf(nbhdQ) >= minPnts then
17                     q.flag ←CORE;
18                     nbhdP ← nbhdQ ∪ nbhdP;
19                 else
20                     q.flag ←BORDER;
21             else if q.flag is NOISE then
22                 q.flag ←BORDER;
23                 q.clusterID ←clusterID;
24     clusterID+ = 1;
```

number of clusters **K**, initial seeds and the dataset. The initial seeds can be randomly generated or can be randomly selected from the dataset. The following are the steps for the algorithm:

- Data assignment step: Each data point is assigned to the closest cluster based on the closest Euclidean distance

- Centroid update step: The centroids are recomputed by calculating the means of the points assigned to that centroid's cluster.

- Iterative step: After calculating the new centroids we can stop at this point if the old centroids is equal to the new centroid which means we have converged or we can iterate a specific number of times and stop at that point. According to [7], it is sure that the results would finally converge, but there is a possibility that we reach the local optimum for the initial seeds and we might get different results with different seeds.

2.3.2 Choosing K

Determining the number of clusters K in a dataset involves applying the algorithm with different values of K and then determining the quality of clusters using comparative index like silhouette coefficient [**?**] and [3].

Chapter 3

Hadoop and MapReduce paradigm

3.1 Apache Hadoop

The Apache Hadoop software library is a framework that is used for the distributed processing of large data sets across clusters of computers using MapReduce paradigm. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. There are four components which form the framework:

- Hadoop Common: contains java libraries and utilities which is required by other modules.

- Hadoop Distributed File System(HDFS): a distributed file system which provides access to application data.

- Hadoop YARN: frame work for job scheduling and cluster management

- Hadoop MapReduce: a YARN based system for parallel processing of data and is the execution engine of the framework.

We are going to limit our discussion to the of details of HDFS and Hadoop MapReduce.

3.1.1 High Level Architecture of Hadoop

Hadoop follows a master slave architecture as illustrated in the Figure 3.1 for parallel processing and data storage which constitutes the MapReduce layer and HDFS respectively. The master node of the HDFS layer is the namenode and the master node of the MapReduce layer is the Job Tracker. The slave nodes form the other systems in the cluster which perform computations and data storage. Each of these systems have a Task Tracker daemon and a DataNode to communicate with the Job Tracker and the NameNode respectively.

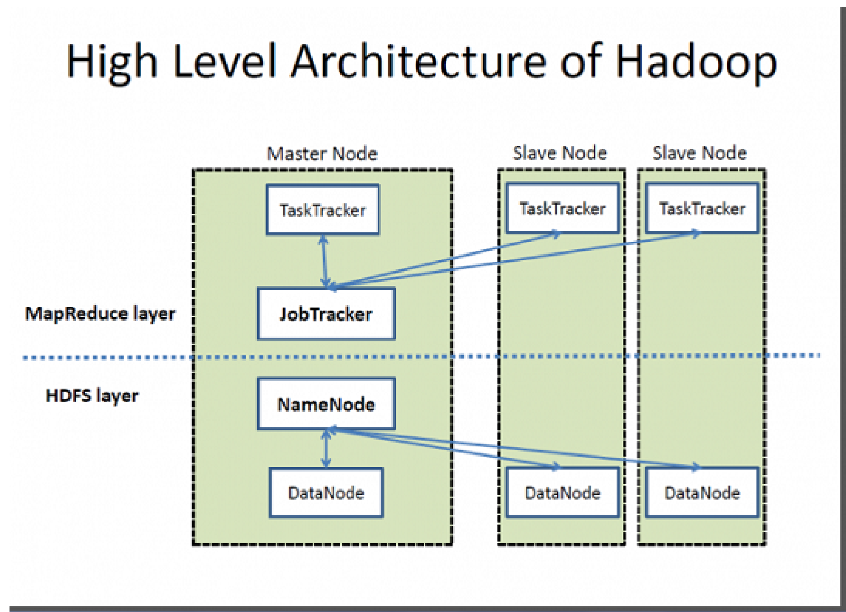


Figure 3.1: Hadoop Architecture

3.1.2 HDFS

HDFS provides access to the data for applications and also stores the data in a distributed fashion. The main purpose of this layer is to make a clear separation of concern from the developers, who only needs to deal with the processing and computation of data. The following are some of the goals/features of the HDFS:

- Data Replication: The data is replicated in more than one node to avoid the loss of data due to hardware/software failure.
- Streaming data access: The emphasis is put on high throughput of data than low latency of data access.
- Large data sets: It can support operations on huge amount of data ranging from gigabytes to terabytes in size.
- Localizing computations: It aims on bringing computations closer to data than the other way around. This lowers network congestion and traffic especially when large amount of data is involved.

HDFS Architecture

HDFS has a master slave architecture with one of the nodes in the cluster acting as the master also known as NameNode and the other usually one per node in the cluster acting as the DataNode, which is also responsible for managing storage in that particular node. HDFS exposes a file system namespace and allows user data to be stored in files. Internally the files are divided into blocks which are usually of size 64MB each and then stored in a distributed manner in a set DataNodes. The NameNode is responsible for file system namespace operations like opening, closing, and renaming files and directories. It also performs the mapping of blocks

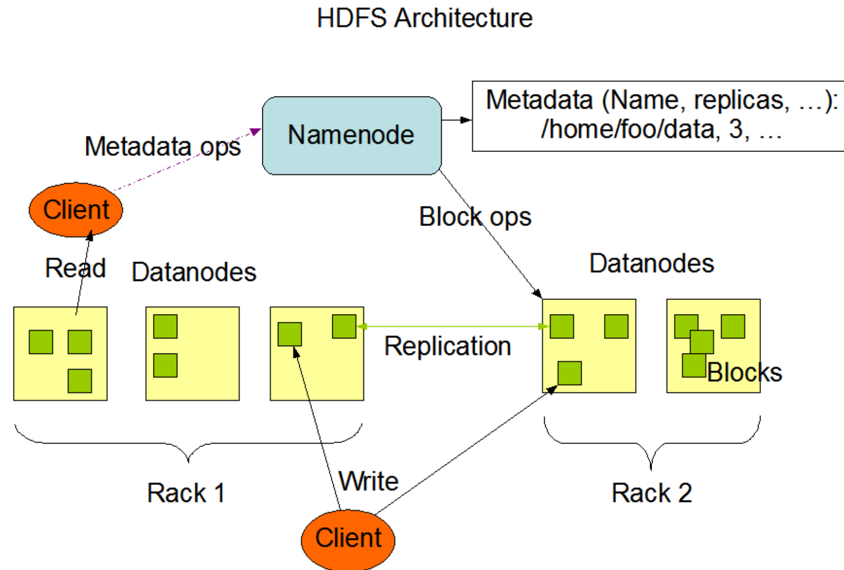


Figure 3.2: HDFS Architecture

to DataNodes. The DataNodes are responsible for serving read and write requests from the file systems clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode. The softwares responsible for NameNode and DataNode are written in Java language and any machine that has a Java environment can be a DataNode/NameNode. Usually in a cluster, one of the machine is assigned the NameNode and the other usually one per cluster is assigned the DataNode.

FileSytem Namespace

HDFS has a traditional hierarchical file organization. A user can create directories and store files inside these directories. NameNode is responsible for the Namespace system and any changes to it is recorded by it.

Data Replication

Fault tolerance is one of the features of the HDFS, it prevents the loss of data blocks due to hardware/software faults in the DataNodes. This is done by replicating the blocks in more than one DataNodes, usually the number is 3. This can be changed by the user.

3.1.3 MapReduce paradigm

MapReduce is processing technique and programming model used for processing large amounts of data in a distributed manner. It has two important parts. The first part is a Map, which process's input and produces key/value pairs. The values from the same key are enumerated to form a new key/List<value> pair which is fed to the second part called the Reducer, which process's it and writes to the HDFS. The beauty of this paradigm is that if any algorithm can be decomposed into map and reduce tasks then it comes scalable in a distributed environment, but the task of decomposing is not trivial.

Hadoop-MapReduce

MapReduce program/application executes in three stages namely map stage, shuffle and sort stage and the reduce stage.

- Map Stage: The mapper's job is to process the input from the HDFS usually line by line to produce key/value pair. On a high level it processes data and produces chunks of small data which needs to be processed.

  Input:k1, v1,Output:list <k2, v2>

- Shuffle and sort stage: Data from the mapper tasks is prepared and moved to the nodes where the reducer tasks will be run. When the mapper task

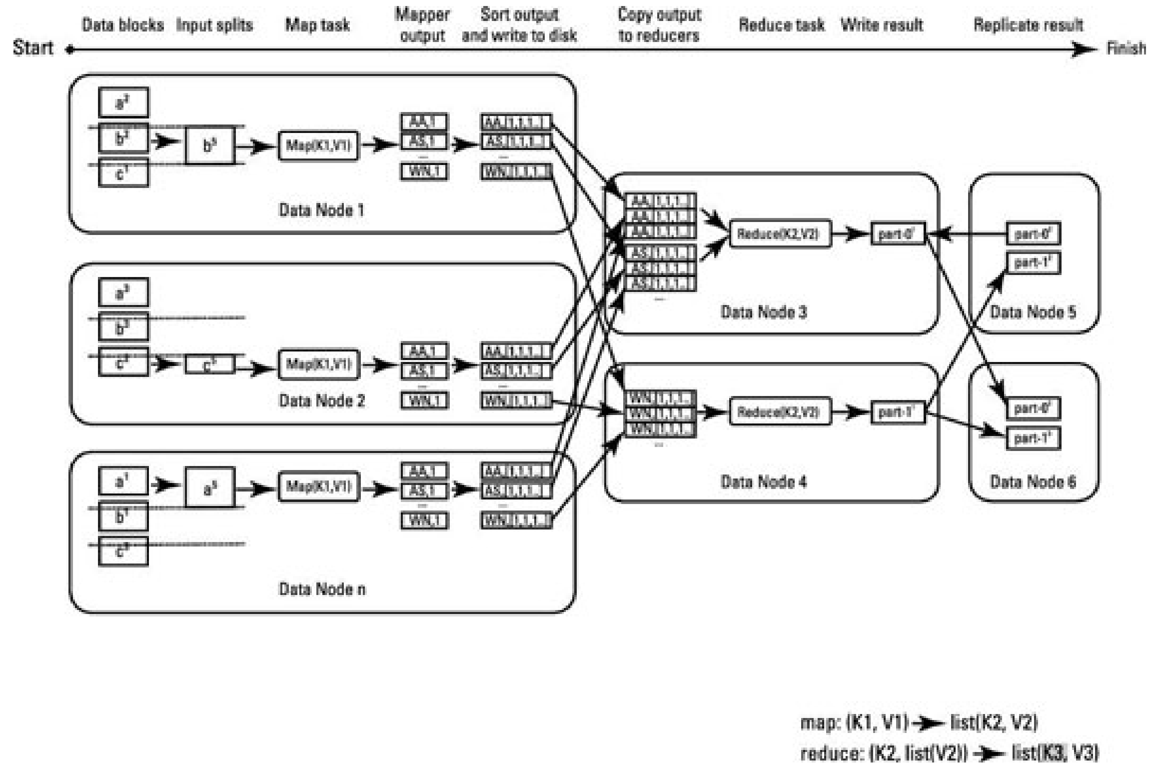


Figure 3.3: Stages in MapReduce

is complete, the results are sorted by key, partitioned if there are multiple reducers, and then written to disk, which is then accessed by reducers.

- Reduce Stage: The reducer's job is to process the values assigned to a key by the mapper and then produce the output which is written to the HDFS.

  Input:k1, List<v> ,Output:k2, v2

The Figure 3.3 illustrates the stages in detail. The framework takes the responsibility of allocating map and reduce tasks to the nodes in cluster, it tries to do it manner that the task is assigned to the nodes where the data resides.

Chapter 4

Map Reduce Implementations

4.1 CURE

The stages described in figure 1.1 needs to be interpreted in terms of Map Reduce paradigm to be implemented in Apache Hadoop. We accomplish this using a single Map Reduce job in Apache Hadoop. The input to the driver program is a csv file where each line is a 2-d point in form of x,y. on a high level, the mapper designates every point its respective partition and the reducer runs the clustering algorithm on the points it is assigned to. The following steps describes the implementation of stages of CURE in Apache Hadoop environment:

- Step 1: Sampling is performed by overriding the getSplits() method of TextInputFormat class. We use reservoir sampling technique to apply sampling on the available splits.

- Step 2:The number of partitions is taken as an input to the program, it is interpreted as the number of reducers in the Map Reduce job. A particular

point is assigned to the respective reducer using the relation:

$$Reducer = key \bmod (Number of Partitions) \tag{4.1}$$

key refers to the line number of the input in the CSV file.

- Step 3: A modified version of CURE is run on points assigned to each of the reducers from step 2. This is described in detail in the following section.

- Step 4: When the number of clusters formed is equal to the 1/3 of the initial number of clusters we started with, in this case it would be the number of points we began with. We would remove the clusters having less than 2 representative points. Once the outliers are removed, we resume with the clustering of the points using CURE. We use this technique which was described in [6].

- Step 5: After the Map Reduce job is completed, we collect all the set of clusters from the reducers and then run the CURE algorithm centrally on this set till we get the required number of clusters. All the stages except sampling is applied on the the result to get the final set of clusters.

4.2 Bottlenecks and Challenges

- According to the algorithm for every merged cluster we have to iterate through the priority queue to update closest clusters and update the priority queue using the closest distance parameter. This contributes to the time complexity by time complexity of O($nlogn$). This makes the brute force approach not scalable especially for the data

- The number of partitions determines the size of the partitions and every par-

tition is processed by a reducer. If the size of the partition is large and since the time complexity is high it is paramount that each reducer is assigned a partition which it can complete processing before it times out.

4.2.1 k-merge Implementation

The time complexity of the CURE algorithm is high since for every two clusters merged the size of the priority queue decreases by 1 and also there is a need to iterate through the remaining clusters in the queue. We can confer that the majority of the time in the algorithm is spent on processing the priority queue. The main idea behind this approach is to pick k closest pair of clusters from the queue and then merge them into one cluster. We found this approach not scalable for large data sets even though it decreases execution time by speeding up the merging process but it does not decrease the time complexity, it remains the same. The flowchart in Figure 4.1 presents a high level view of the steps involved in this implementation.

4.2.2 Hash Map Implementation

The time complexity of the CURE algorithm can be reduced by applying an approximation technique so that for every two clusters merged, we only iterate through the subset of remaining clusters in the queue. Using this approach, we use a Hash Map where key is the cluster id and the values are the clusters it is closest to. Suppose u and v are the clusters getting merged to form a cluster w, we could use the hash map to determine the the clusters which are going to get affected by the merge and then use the kd tree to update their closest clusters,thereby avoiding to iterate through the each of the clusters in the priority queue. Also after determining the closest cluster x to the merged cluster w, there is a chance that closest cluster to cluster x

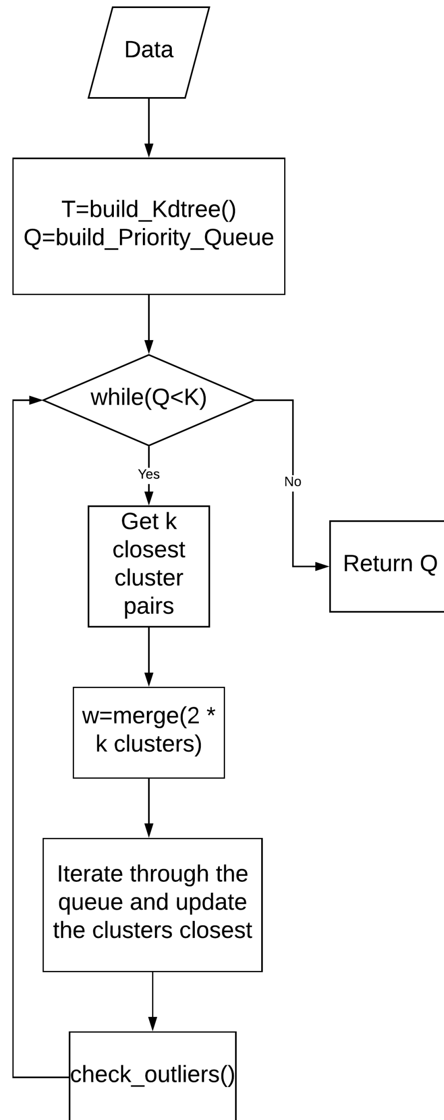


Figure 4.1: k-merge implementation of CURE

might get affected, therefore we update the closest cluster to cluster x. Using this method there is a possibility that we might not have updated all the clusters which have the merged cluster w as its closest cluster, after removing closest clusters u and v from the priority queue we do a check whether v is indeed the closest cluster to u and vice versa there by mitigating its effect. The following is the pseudocode of this approach :

```
Cure_HashMap(int k,int c,PriorityQueue<Cluster> Q,kdtree T)
```

```
        //Initialize HashMap
        Map<Cluster,List<Cluster>> hmap=new HashMap();
        //Outlier condition check
        int num_points_outliers=Q.size() * 1/3;
        int check_outlier=false;
        while(Q.size()<k){
        Cluster u=Q.poll();
        cluster v=u.getClosest();
        cluster u_closest=getClosestCluster(T,u);
        while(v!=u_closest){
                updateHashMap();
                if(v.closest==u){
                Cluster v_closest=getClosestCluster(T,v);
                        updateHashMap();
                }
                Q.add(u);
                u=Q.poll();
                v=u.getClosest();
                u_closest=getClosestCluster(T,u);
        }
        //delete v from priority Queue
        Q.remove(v);
        //delete u and v's representative points
        T.delNode(u.getRep());
        T.delNode(u.getRep());
        Cluster w=merge(u,v,c,alpha);
        //add w's points to kdtree
        T.insertNode(w.getRep());
        //modify the hash map and
get the list of modified clusters from hashmap
        List<Cluster> modified_clusters=new ArrayList<>();
        updateHashMap(modified_clusters);
        Update the clusters in modified_clusters
        Get the closest cluster to w
        cluster w_closest=getClosestCluster(T,w);
        Get the clusters closest to w_closest and
do a check whether its getting modified
        //set the closest to w_closest
        w.setClosest(w_closest);
        //add w to Queue
        Q.add(w);
        //check for outlier condition
        if(num_points_outliers<=Q.size()
&&!checkOutlier){
                checkOutlier=True;
                removeOutliers();
        }

}
```

The flowchart in Figure 4.2 presents a high level view of the steps involved in this implementation.

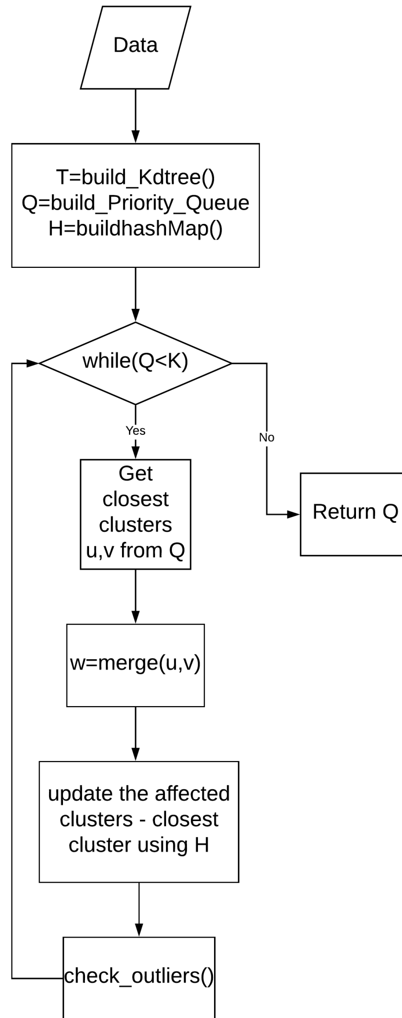


Figure 4.2: hashMap implementation of CURE

The time complexity of this approach is O($nlogn$). Even though this approach brings down the time complexity but it will have a impact the quality of clusters produced. In the following section we evaluate the quality of clusters produced using this approach.

4.3 DBSCAN

DBSCAN is a density based clustering algorithm and we are going to implement the version proposed by [8] and then determine the bottleneck/challenges of the approach. There are four stages involved in the process as shown in the 4.3 and

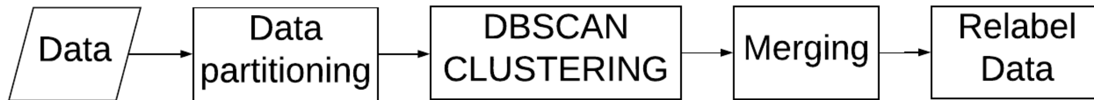


Figure 4.3: Stages in DBSCAN

are described in high level below:

- Data Partitioning: We determine the appropriate partitions for the input data using a cost based approach.
- Clustering: For each partition we run the DBSCAN clustering algorithm for the points belonging to the partition.
- Merging: We perform the intersection of the results from the last stage and determine the points which simultaneously belong to different partitions. The cluster ids of these points are used to perform mapping from local cluster ids to global cluster ids.
- Relabeling: We use the mapping of the global cluster id to local cluster id from the last stage to relabel the cluster id's which have been merged.

4.3.1 Data Partitioning

The different steps involved in this stage are shown in the Figure 4.4.

The goal of this stage is to determine partitions of the data in such a way that we can run clustering task on each of them independently. Initially we consider the the

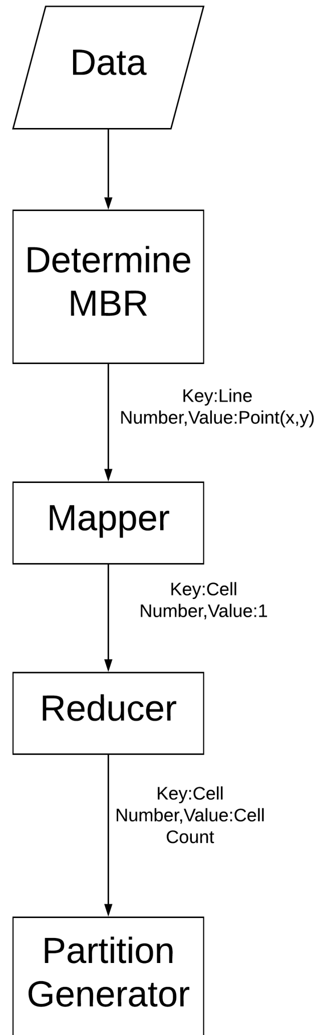


Figure 4.4: Steps in Data Partitioning

whole input to be contained within a rectangle which we refer to as MBR(Minimum Bounding Rectangle),S_U. We then divide S_U into cells whose side length is $2 * \epsilon$. The MBR is going to be covering a large area if the data is taken in as it is. There is a need to normalize the data so that the MBR is small enough that it can be processed. If the values in the data under consideration is positive we can normalize the data between [0,1] otherwise we normalize between [-1,1].

Data coordinates are *normalized* to [0,1] so that the MBR has a lower left bottom

corner of (0,0) and a top right corner of (1,1) if we consider the case where the values are positive . A MapReduce job is started to determine the the number of points inside each cell. The output from the job is fed to the partition generator which is a centralized process.

Partition Generation

This step is performed in a centralized fashion. The input to this step is the output from the MapReduce job. The output contains the count of the data points contained under each cell.

We proceed by partitioning S_U into two sub rectangles by considering splitlines along the cells. Each splitline divides the main rectangle into 2 sub rectangles, and then cost is assigned to each sub rectangle. The cost of the rectangle is enumeration of the cost of each cell and is calculated the the equation.

$$Cost(C_i) = 1 + h + \sqrt{Nc_i} * \frac{1}{\sqrt{f} - 1} + Nc_i * \frac{1}{f - 1} \tag{4.2}$$

$$h = 1 + \lceil \frac{\log \frac{N_S}{f}}{\log f} \rceil \tag{4.3}$$

where C_i is the cost of cell i, Nc_i is the number of a points contained inside cell i,f is the fanout of the R-tree and N_S is the total number of points contained inside rectangle S. The splitline selected will have the minimum cost difference for the sub rectangles created. The algorithms 5, 6 and 7 explains the step in detail. once the partitions are generated, we determine the intersecting partitions which is later used in the merging stage. The following Figure 4.5 illustrates the process in a high level.

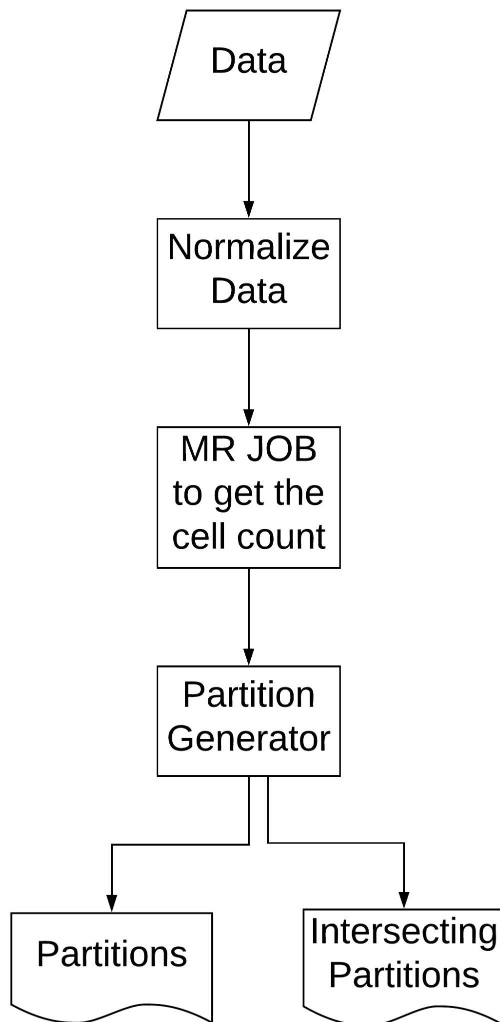


Figure 4.5: Partitioning overview

4.3.2 DBSCAN Clustering

This phase involves a MapReduce job to perform the clustering of each partition using DBSCAN. The mapper allocates every point to its corresponding partition. Before allocating every point it expands the partition by ϵ . This is done to determine the border points which will be later used to merge the clusters which belong in different partitions but constitute a single cluster on their own. There is a reducer started for every partition that had been generated. Reducer receives

the points allocated to a partition and performs clustering. It writes output to three files namely file-results-partitionID, file-AP-partitionID, file-BP-partitionID. The file-results-partitionID contains the overall results of clustering i.e the points along with the clusterID' s and the type. The file-AP-partitionID file contains points which belong to the inner margin of the partition and also is a core point. The file-BP-partitionID file contains points which belong to the outer margin of the partition and also is not a noise point. These files are later used in the merge phase.

4.3.3 Merge Phase

We use the fast merge theorem stated and proved in [8] to determine the merge points. The theorem states the following :

$$Mergepoints = (AP_1 \cap BP_2) \cup (AP_2 \cap BP_1) \tag{4.4}$$

where 1 and 2 are intersecting partitions, AP and BP represents files file-AP-partitionID and file-BP-partitionID for respective partitions. This phase represents a MapReduce job where the input to the mapper is the file containing the intersecting partitions, which we had determined in the first stage. We then determine the merge points using (4.4), the clusterID's of the merge points are sent to one reducer which means we are using the same key. For example, if there is a point p which has clusterId of A and B as it is part of merge points, then the mapper output will have the value (A,B) and a key of NULL. The key is NULL so that it goes to a single reducer. In the reducer we use breadth first search to determine the connected components. The clusterID's belonging to the same component are assigned a global clusterID. The output of the reducer will contain the mapping from the local cluster ID to the global clusterID.

4.3.4 Relabel Data

This final phase also represents a MapReduce job. The input to the mapper is the local clustering results contained in file-results-partitionID(one for each partition). The mapper maps the local cluster ID to the global cluster ID for a point and followed by emitting the point as key and a pair as value which contains the final cluster ID and type of the point. For a point belonging in the margins of a partition it might belong to more than one partition and hence there might be duplicate keys emitted from the mapper, that means a point might be associated with different types. The values of the duplicate key are processed in the reducer. In the reducer, we assign a type(BORDER,NOISE,CORE) to the point also referred as the duplicate key here, based on the following priority in the order of high to low :CORE,BORDER and NOISE. The unique keys from the mapper are directly written to the output.

Algorithm 5: Cost-based spatial partitioning

Result: partitions
Input : S_U,nPointsOfCell:Array containing number of points in each cell,maxCost:Maximum cost for each partition
Output: Set of non-overlapping partitions

```
1  taskQueue ← S_U ;
2  partition ← emptyset;
3  while taskQueue is not empty do
4      S ←pop from taskQueue ;
5      if EstimateCost(S, nPointsOfCell) > maxCost then
6          (S_1, S_2) ← costBasedBinarySplit(S,nPointsOfCell);
7          push (S_1, S_2) into taskQueue ;
8      else
9          add S to partitions ;
10 return partitions;
```

Algorithm 6: Cost based binary split

Result: S_1, S_2
Input : S:the rectangle to split,nPointsOfCell
Output: sub rectangles S_1, S_2

1 $(S_1, S_2) \leftarrow$(NULL,NULL) ;
2 $minDiff \leftarrow \infty$;
3 *splitLineCandidates* ←all vertical and horizontal lines that split S into sub rectangles;
4 **foreach** *splitLine in splitLineCandidates* **do**
5 $(R_1, R_2) \leftarrow$ sub rectangles corresponding to splitLine ;
6 $R_1.cost \leftarrow$ EstimateCost(R_1,nPointsOfCell);
7 $R_2.cost \leftarrow$ EstimateCost(R_2,nPointsOfCell);
8 $costDiff \leftarrow |R_1.cost - R_2.cost|$;
9 **if** $costDiff < minDiff$ **then**
10 $minDiff \leftarrow costDiff$;
11 $(S_1, S_2) \leftarrow (R_1, R_2)$;
12 return (S_1, S_2);

Algorithm 7: Estimate Cost

Result: cost
Input : S,n
Output: cost

1 **foreach** *cell in S* **do**
2 $nPoints$ ←get the number of points in cell;
3 $costOfCell$ ←use equation (4.2) ;
4 return cost ;

4.3.5 Optimization

The number of cells will be increasing as the value of the epsilon decreases. For example: the ϵ value of 0.0001 will be creating a MBR of 10^8 cells. Consider the function call EstimateCost() in line number 5 of Algorithm 5, there is a possibility that at one point of time the enumerated cost of processed cells might be higher than the maxCost parameter of Algorithm 5 and the processing of the rest of the cells is not necessary. One way to avoid this is to pass in the maxCost parameter to the function EstimateCost() and to return the cost at the point where the cost

of processed cell is more than the maxCost parameter thereby avoiding the need to process all the cells. The algorithm 8 illustrates the idea presented here.

Algorithm 8: Improved Estimate Cost

```
   Result: cost
   Input  : S,n,maxcost
   Output: cost
1  foreach cell in S do
2      nPoints ←get the number of points in cell;
3      costOfCell ←use equation (4.2) ;
4      cost+ ← costOfCell;
5      if cost > maxCost then
6          break;
```

4.3.6 Objective

The MapReduce version of DBSCAN can run successfully in a distributed environment like Hadoop as long as the partitions produced are not large enough that it does cause time out of the reducer to which it is assigned to unless reducer time out value is changed or the some partitions are large in size that it makes it processing sequential than parallel. The size of the partition is dependent on two parameters ϵ and maxCost. The feasible values for ϵ and maxCost can be determined by firstly running a MapReduce job to determine the number of points in each cell, followed by feeding the output to the partition generator which gives out the partitions. Secondly we start a new MapReduce job which uses the partitions we generated to do clustering, and if we do not encounter any failed reducers which means we have used the feasible values for ϵ and maxCost. So to determine the feasible values for ϵ and maxCost we require a MapReduce job to determine the cell count for a value of ϵ and which is followed by the partition generator phase and a MapReduce job for clustering for different values of maxCost . Our objective is to explore the relation-

ship between ϵ and maxCost and also look for observations which would shorten the process required to find the feasible values.

4.4 kmeans

The MapReduce implementation is pretty straight forward. The driver acts like controller in the sense that it controls the number of iterations and also loads the latest seeds to the distributed cache so that it can be accessed in the mapper and reducer. The mapper for every data point determines the closest seed and the reducer calculates the new seeds.

- Mapper: The mapper takes a data point as the input and outputs the closest seed as the key and the point along with value of 1 as its value.

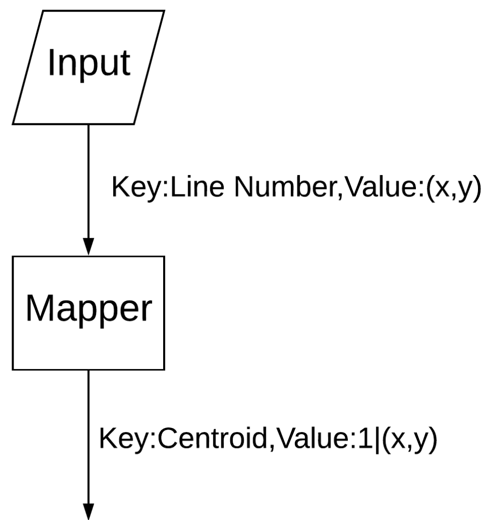


Figure 4.6: Mapper in kmeans

- Reducer:The reducer is run for every seed, and calculates the new seed as it process all the points closest to that particular seed and it is written to the

output. It also determines whether the new centroid calculated is equal to the previous one, the value written is used by the driver. The Figure 4.7 illustrates the function of reducer.

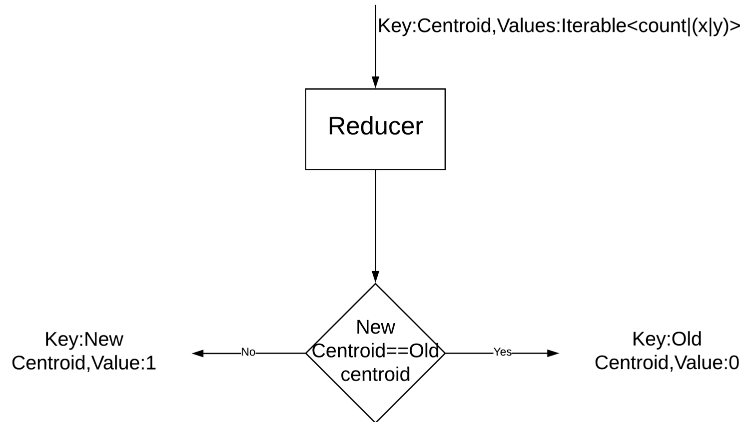


Figure 4.7: Reducer in kmeans

- Driver: The driver process's the output from the reducer and checks whether the recently calculated seeds is similar to the previous seeds, if so it would terminate the program as the clustering is completed else it would start a new MapReduce job to calculate the seeds. It would do so until the number of iterations is equal to the input the user has provided.

4.4.1 Optimization

We see that for every new iteration started the input is loaded and processed before being assigned to the mapper. One thought that comes to the mind is to load the input into the distributed cache and use it for further processing. But the problem with that approach is that the distributed cache has a size limit of 10GB, but often input size will exceed that limit making that approach unfeasible. Another approach is to use a combiner in addition to a mapper and reducer. Combiner's primary job

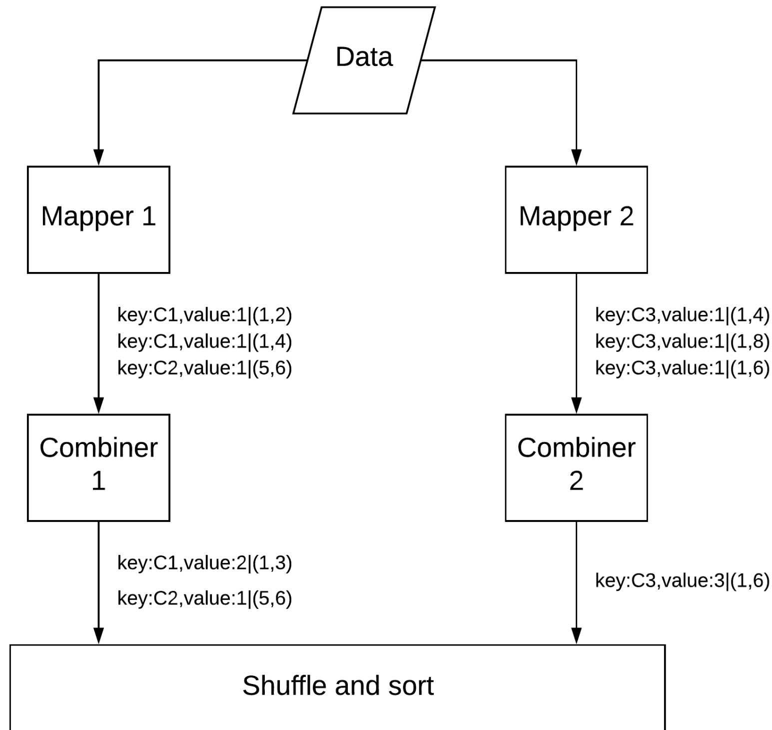


Figure 4.8: Use of Combiner

is to summarize the output from the mapper, and it usually runs after the mapper and before the reducer. It works on the intermediate key/value pairs and reduces the number of key/value pairs before sending to the reducer there by decreasing the network bandwidth and congestion. The output from the mapper which consists of the seed and the points assigned to it. The combiner uses the output from the mapper and calculates the intermediate mean and then outputs the seed as the key and the value consists of the intermediate mean along with count of the points that had been assigned. The count is also written to the output because it is needed in the reducer to calculate the new seed, as the product of the intermediate mean and the count will give us the respective sums to calculate the new seed. This optimization is trivial and straight forward. The goal is to understand the its effect on the performance of the clustering. We would run it against set of varying

inputs and evaluate its performance with and without combiner which would help us understand the effect of network traffic on completion of a MapReduce job.

Chapter 5

Experiments and Evaluation

5.1 CURE

5.1.1 Experimental Setup

For evaluation purpose, we have created a cluster using Amazon EMR having 1 master node and 2 slave nodes and having a software configuration of Hadoop 2.8.3. Each of the nodes have a configuration of 4v CPU and 8 GB memory.

5.1.2 Datasets

We use 2 datasets, the first one is a custom dataset and the second one is a real world dataset. The following describes the dataset in detail.

- Custom dataset:The input is a set of points in the form (x,y) which is created randomly and is not a real world dataset. The size of the dataset varies between 2.3 MB and 1.1 GB.

- NYC taxi and limousine trip dataset: This dataset represents the trips made by yellow and green taxi's in the New York city between the year 2014 to

2017. The trip records include fields capturing pick-up and drop-off dates/-times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts. The two fields we use for clustering are trip distance and total fare amount. The dataset has size of 71.9 GB.

5.1.3 Input parameters

- k=300(MapReduce job) and k=10(Centralized Processing)
- c(Number of representative points)=56
- Shrinking factor=0.8

5.1.4 Results and Evaluation

We test the dataset of varying sizes, we then record the number of partitions and the processing time against varying sample rates. The goal is to evaluate the quality of the clusters produced for different sampling rates and also understand the number of partitions required to do so. We determine the accuracy using Silhouette coefficient, It is found to be best performing index in comparative experiments according to [12] and [3]. The value of the Silhouette coefficient varies between [-1,1], the closer the value is to 1 the better is the quality of the clusters and a negative value indicates that the produced clusters are not distinct enough. We have also calculated the Silhouette coefficient for the dataset we are testing against using a sequential clustering algorithm called "Basic Sequential Algorithmic Scheme " also known as "BSAS". It is a linear clustering algorithm and we have used it as a baseline to make a comparison between the coefficient's obtained from the modified version of CURE

Size(MB)	Time(hh:mm)	Partitions	Accuracy
2.3	00:01	10	0.338746
23.3	00:05	10	0.310756
231	00:13	10	0.27502496
1126.4	00:14	10	0.22284046

Table 5.1: Sample rate:1%

Size(MB)	Time(hh:mm)	Partitions	Accuracy
2.3	00:01	10	0.338746
23.3	00:05	10	0.310756
231	00:13	10	0.258733124
1126.4	00:13	10	0.271897678

Table 5.2: Sample rate:5%

which uses a hash map and the linear sequential algorithm. The input parameters to the BSAS are the threshold of dissimilarity which is assigned a value of 100 and the number of maximum clusters q which assigned a value of 10.

5.1.5 Custom Dataset

The following Tables 5.1,5.2,5.3,5.4,5.5 below show the results for the different sampling rates and input sizes for the modified version of CURE. The Table 5.6 shows the coefficient's calculated for the same dataset using BSAS.

Size(MB)	Time(hh:mm)	Partitions	Accuracy
2.3	00:01	10	0.338746
23.3	00:05	10	0.310756
231	00:06	10	0.3087494
1126.4	00:38	10	0.2997695

Table 5.3: Sample rate:10%

Size(MB)	Time(hh:mm)	Partitions	Accuracy
2.3	00:01	10	0.338746
23.3	00:05	10	0.310756
231	00:06	10	0.3087494
1126.4	00:51	30	0.30764198

Table 5.4: Sample rate:20%

Size(MB)	Time(hh:mm)	Partitions	Accuracy
2.3	00:01	10	0.338746
23.3	00:05	10	0.310756
231	00:33	10	0.283745
1126.4	02:56	1000	0.3105107

Table 5.5: Sample rate:60%

Size(MB)	Accuracy
2.3	0.475721
23.3	0.469697
231	0.477696
1126.4	0.471976

Table 5.6: Silhouette coefficient calculated for sequential clustering

Evaluation

We can infer from the tables above that as we increase the sample rates the accuracy of the clusters produced is increasing particularly for dataset with size 1126.4 MB which seems conceivable. But in comparison to coefficient's calculated for the dataset using sequential clustering in Table 5.6, the values are lower which indicates that the quality of clusters produced in the modified version of CURE is not good as the one's in the sequential clustering. We can also observe that as the sampling rate increases particularly for the larger dataset of sizes 231 MB and 1.1 GB , we are required to create more number of partitions(reducers) for successfully clustering. The value of number of partitions was chosen based on the notion that each reducer were assigned points that it can handle before it time outs, but large values of reducers without taking in account of the degree of parallelization makes the computation more sequential than parallel.

5.1.6 Real World Dataset

We run the algorithm against the NYC taxi and limousine trip dataset using different sample rates and number of partitions to observe the run time and the accuracy of the clusters produced. The following table

Sampling rate(%)	Time(hh:mm)	Partitions	Accuracy
1	01:24	1100	-0.35811067768
5	03:09	1650	-0.1737438564
10	06:10	3204	-0.199098639

Table 5.7: Real World Dataset Results

Evaluation

We observe that the the accuracy increases when the sampling rate increases from 1% to 5%, but the accuracy decreases when the sampling rate increases from 5% to 10%. We observe that the coefficient for the same dataset using BSAS clustering algorithm is equal to 0.7286294251401926, in comparison to values in Table 5.7 for different sample rates. We can infer that the quality of clusters produced is poor in comparison to the "BSAS", one value is closer to 1 and the other is negative. This can be attributed to the fact that the dataset contains data from the years 2014-2017, and the output from sampling may not contain data from all the years.

5.2 DBSCAN

Our objective is to explore the relationship between ϵ and maxCost and also look for observations which would shorten the process required to find the feasible value for the above parameters. We would do so by observing the number of failed reducers, the number of partitions and average size of these partitions for different values of ϵ and maxCost.

5.2.1 Experimental Setup

For evaluation purpose, we have created a cluster using Amazon EMR having 1 master node and 2 slave nodes and having a software configuration of Hadoop 2.8.3. Each of the nodes have a configuration of 4v CPU and 8 GB memory.

5.2.2 Dataset

NYC taxi and limousine trip dataset: This dataset represents the trips made by yellow and green taxi's in the New York city. I have taken the dataset for the year

2014 and for the month January which is of the size 2.16 GB but after normalization the size comes up to 397 MB. It contains 9657376 records.

5.2.3 Results and evaluation

maxCost	Number of partitions	Number of reducers	Failed	Average Size(Points/partition)
100000	1	1	1	9657376
10000	3	3	1	3219125.3
1000	3	3	1	3219125.3

Table 5.8: *epsilon*:0.1

maxCost	Number of partitions	Number of reducers	Failed	Average Size(Points/partition)
100000	2	2	2	4828688
10000	5	5	3	1931475.2
1000	15	15	12	643825.066667

Table 5.9: *epsilon*:0.01

maxCost	Number of partitions	Number of reducers	Failed	Average Size(Points/partition)
100000	2	2	2	4828688
10000	19	19	15	508282
1000	112	112	20	86227

Table 5.10: *epsilon*:0.001

maxCost	Number of partitions	Number of reducers	Failed	Average Size(Points/partition)
1000	8	8	0	1207172
10000	64	64	0	150896

Table 5.11: *epsilon*:0.0001

5.2.4 Evaluation

We can make the following observations from the above tables which would help one to decide on the values of maxCost and ϵ:

- If for a particular value of ϵ, if the number of partitions remain the same for different maxCost values it means that epsilon is not small enough, as you can see in Table 5.8
- The number of partitions does not indicate whether the clustering will be done successfully. As you can observe that in Table 5.11 the clustering is done successfully with fewer partitions than in Table 5.10. This happens because the points are unevenly distributed with few partitions having high number of points and few having very few compared to the other.
- The lower the value of epsilon the better the chance of successful clustering, because every partition would assign fewer number of points from the partitions which are intersecting with it.

5.3 k-means

The optimization applied to the k-means is the use of a combiner. The combiner acts like a "mini reducer" and summarizes the output from a mapper thereby decreasing the network congestion and traffic. The objective here is to determine the effect of combiner on the run time of the k-means MapReduce job and to accomplish this we would run the clustering with/without combiner on same set of input and record the run time for a single iteration.

5.3.1 Set up

All experiments were conducted on a cluster consisting of 2 nodes. Each node consists of 56 Intel@Xenon CPU E5-2690 2.60GHz processors. 500GB RAM, 7TB HDD. Each node is connected to a Gigabyte Ethernet switch and runs Ubuntu 16.04.3 LTS with Spark-2.02 and Hadoop-2.73. The input to the k-means clustering job consisted of a text file consisting of 3 seeds.

5.3.2 Dataset

NYC taxi and limousine trip dataset: This dataset represents the trips made by yellow and green taxi's in the New York city between the year 2013 to 2017. The trip records include fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts. The two fields we use for clustering are trip distance and total fare amount. The dataset has size of 98.9 GB.

5.3.3 Results

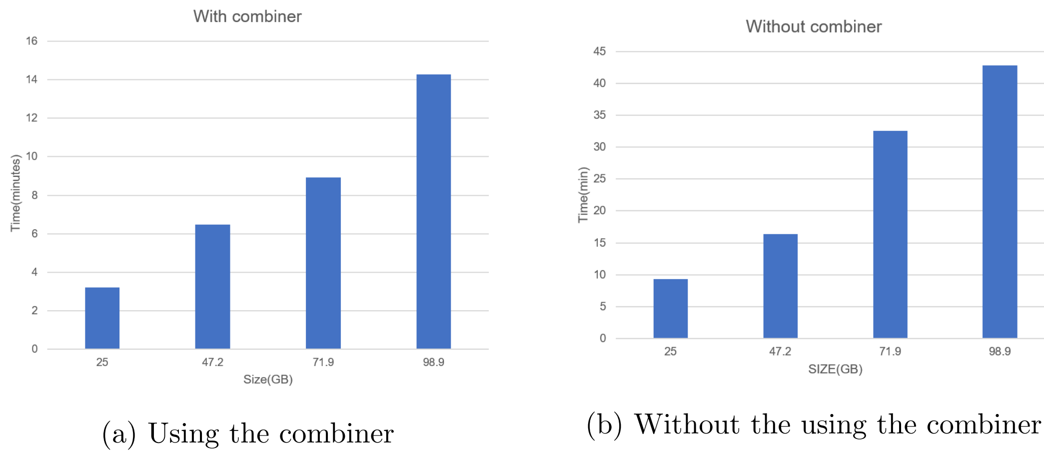
The following tables demonstrates our observation of the run time of the single iteration of the k-means clustering job with/without using combiner:

Size(GB)	Time(mm:ss)
25	03:13
47.2	06:28
71.9	08:55
98.9	14:16

Table 5.12: Using the combiner

Size(GB)	Time(mm:ss)
25	09:16
47.2	16:22
71.9	32:32
98.9	42:51

Table 5.13: Without using the combiner



(a) Using the combiner

(b) Without the using the combiner

Figure 5.1: Results for K- Means

5.3.4 Evaluation

On comparison of the results we can infer that the k-means clustering job with a combiner takes almost 1/3rd run time of the one without a combiner. This highlights the significance of the use of a combiner on the k-means clustering job.

Chapter 6

Related Work

6.1 CURE

A MapReduce version of CURE is proposed by [10], they have performed all the stages involved in CURE in a single Reducer as it is designed for a single node cluster. It is not certain that their approach is scalable as the largest test dataset consists of 10000 points, which is not realistic in the environment where typically datasets are huge.

6.2 K-Means

There is a similar comparative analysis done by [1] where they perform K-Means clustering with/without combiner but the largest dataset had 5 million records which is not realistic in the environment where typically datasets are huge.

Chapter 7

Conclusion and Future work

7.1 CURE

The distributed version of CURE proposed which uses a hash map to reduce the time complexity from O($n^2 logn$) to O($nlogn$) by applying an approximation is scalable, as it can be observed in the results section that it can handle large dataset but the quality of clusters seems to be compromised especially in comparison to the clusters produced by "BSAS". We can conclude that the approach is scalable as it can handle large datasets but since the quality of clusters produced are not up to the standard in comparison to the ones produced by "BSAS", this method cannot be used practically

The value of number of partitions was chosen based on the notion that each reducer were assigned points that it can handle before it time outs, but large values of reducers without taking in account of the degree of parallelization makes the computation more sequential than parallel. Usually the number of reducers are

assigned by using relation

$$Maximum\, Number\, of\, reducers = (degree\, of\, parallelization) * 10 \tag{7.1}$$

Future work would include determining the performance of the approach using the (7.1) to determine the number of reducers that can be used. The timeout value of the reducer can be manipulated by changing the value of mapred.task.timeout in mapred-site.xml, by default it is 600000 milliseconds but it requires root user privileges to change its value.

7.2 DBSCAN

We explored the relation between parameters in the MapReduce version of DBSCAN: ϵ and maxCost(maximum cost of a partition), to make observations which could help one decide the values for them. We made three observations. Firstly, we observed that for a particular value of ϵ, if the number of partitions remain the same for different maxCost values we could infer that ϵ is not small enough. Secondly, we observed that the large number of partitions does not indicate successful partition generation, there is the possibility that there is an uneven distribution of points for the partitions. Finally we observed that the lower the value of epsilon the better is the chance of successful clustering, because every partition would assign fewer number of points from the partitions which are intersecting with it.

We also proposed an optimization to the algorithm 7 which is used to estimate cost for a rectangle S, the main idea proposed is not to iterate through all the cells of the rectangle if the enumerated cost is higher than the maxCost parameter specified. This would help to reduce the runtime when the ϵ is low and there are large number of cells to be processed.

Future work would include to use Silhouette coefficient to determine the quality of the clusters produced for different values of epsilon in DBSCAN and make a comparison to the one's produced by the "BSAS" algorithm.

7.3 k-means

The use of combiner is an optimization that can be applied to the MapReduce version of k-means, we wanted to determine its effect on the performance. We measured the runtime of single iteration of clustering by using MapReduce version of k-means with/without combiner against dataset of various sizes. On comparison of the results we could infer that the k-means clustering job with a combiner takes almost 1/3rd run time of the one without a combiner, which highlighted the significance of a combiner.

Appendix A

Basic sequential algorithmic scheme

The basic sequential algorithmic scheme (BSAS) is a sequential clustering algorithm and is linear in order. It makes a single pass through the data points and the number of clusters is not known before. The inputs to the algorithm are threshold of dissimilarity T(maximum distance allowed between a point and existing cluster) and the number of maximum clusters allowed q.

Algorithm 9: BSAS

Result: w
Input : set of points X,threshold distance T,maximum number of clusters allowed
Output: C:List of clusters

1 $m \leftarrow 1$;
2 C_1 ←initialize the first cluster with the first point ;
3 **foreach** $x \; in \; X$ **do**
4 $\quad C_k \leftarrow$ closest cluster to x;
5 $\quad$ **if** $d(x, Ck) > TAND(m < q)$ **then**
6 $\quad\quad m+ \leftarrow 1$;
7 $\quad\quad C_m \leftarrow$ create new cluster containing x;
8 $\quad$ Else add x to C_k;

Bibliography

[1] Prajesh P Anchalia. Improved mapreduce k-means clustering algorithm with combiner. In *Computer Modelling and Simulation (UKSim), 2014 UKSim-AMSS 16th International Conference on*, pages 386–391. IEEE, 2014.

[2] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. In *ACM Sigmod record*, volume 28, pages 49–60. ACM, 1999.

[3] Olatz Arbelaitz, Ibai Gurrutxaga, Javier Muguerza, JesúS M PéRez, and IñIgo Perona. An extensive comparative study of cluster validity indices. *Pattern Recognition*, 46(1):243–256, 2013.

[4] Dorin Comaniciu and Peter Meer. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on pattern analysis and machine intelligence*, 24(5):603–619, 2002.

[5] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[6] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Cure: an efficient clustering algorithm for large databases. In *ACM Sigmod Record*, volume 27, pages 73–84. ACM, 1998.

[7] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[8] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. Mr-dbscan: a scalable mapreduce-based dbscan algorithm for heavily skewed data. *Frontiers of Computer Science*, 8(1):83–99, 2014.

[9] George Karypis, Eui-Hong Han, and Vipin Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.

[10] Piyush Lathiya and Rinkle Rani. Improved cure clustering for big data using hadoop and mapreduce. In *Inventive Computation Technologies (ICICT), International Conference on*, volume 3, pages 1–5. IEEE, 2016.

[11] Raymond T. Ng and Jiawei Han. Clarans: A method for clustering objects for spatial data mining. *IEEE transactions on knowledge and data engineering*, 14(5):1003–1016, 2002.

[12] Lucas Vendramin, Ricardo JGB Campello, and Eduardo R Hruschka. Relative clustering validity criteria: A comparative overview. *Statistical analysis and data mining: the ASA data science journal*, 3(4):209–235, 2010.

[13] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: an efficient data clustering method for very large databases. In *ACM Sigmod Record*, volume 25, pages 103–114. ACM, 1996.