Parallelization of Hierarchical Density-Based Clustering using MapReduce

by

Talat Iqbal Syed

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Cluster analysis plays a very important role for understanding various phenomena about data without any prior knowledge. However, hierarchical clustering algorithms, which are widely used for its representation of data, are computationally expensive. Recently large datasets are prevalent in many scientific domains but the property of data dependency in a hierarchical clustering method makes it difficult to parallelize. We introduce two parallel algorithms for a density-based hierarchical clustering algorithm, HDBSCAN*. The first method called Random Blocks Approach, based on the parallelization of Single Linkage algorithm, computes an exact hierarchy of HDBSCAN* in parallel while the second method, the Recursive Sampling Approach, computes an approximate version of HDBSCAN* in parallel. To improve the accuracy of the Recursive Sampling Approach, we combine it with a data summarization technique called Data Bubbles. We also provide a method to extract clusters at distributed nodes and form an approximate cluster tree without traversing the complete hierarchy. The algorithms are implemented using the MapReduce Framework and results are evaluated in terms of both accuracy and speed on various datasets.

To my Supervisor, Prof. Jörg Sander

For being the best teacher, mentor and colleague throughout my graduate school.

Acknowledgements

I would like to express my deepest appreciation to my supervisor, Prof. Jörg Sander, without whom, this dissertation would not have been possible. He is a great source of inspiration and encouragement. His contructive feedback and invaluable guidance has always helped me focus on research.

I would also like to thank Prof. Ricardo J. G. B. Campello, Dr. Arthur Zimek, Dr. Davoud Moulavi and Jeeva Paudel with whom I had some of the best discussions and for their useful feedback.

I would also like to express my gratitude to the committee members, Dr. Osmar R. Zaïane and Dr. Eleni Stroulia for their suggestions to improve the dissertation.

Table of Contents

List of Figures

Chapter 1

Introduction

Clustering is an unsupervised learning task that aims at decomposing a dataset into distinct groups called *"clusters"*. These clusters, simplify the representation of the complete dataset and help the user to understand the dataset better. Although there is no universally agreed definition of a cluster [2], many consider a group to be a cluster by considering the internal cohesion called *"homogeneity"* and the external isolation called *"separation"* [3] [4], i.e., patterns in the same cluster should be similar while the patterns from different clusters should be relatively different. A cluster can also defined as "densely connected regions in a multi-dimensional space separated by loosely connected points" [4].

Clustering techniques play an important role in many applications. Clustering is an unsupervised learning task since prior information about data is not known. Traditionally, clustering algorithms are broadly divided into two types:

1. Partitional Clustering
2. Hierarchical Clustering

Given a set of input data $X = \{x_1, x_2, ..., x_n\}$, a partitional clustering algorithm divides the entire dataset into k groups or clusters, where the cluster set is represented as $C = \{C_1, ..., C_k\}$ such that $C_i \neq \emptyset$ and $\bigcup_{i=1}^{k} C_i = \{X\}$. The membership of data objects to a cluster can be both hard, where $C_i \cap C_j = \emptyset; i \neq j$ or fuzzy [5], where each pattern has a variable degree of membership in each of the output clusters. The Hierarchical clustering [6] on the other hand, generates tree-like nested partitions of the set X given by $H = \{H_1, ..., H_m\}; m \leq n$, such that if $C_i \in H_p$, $C_j \in H_q$ and $p > q$ then $C_j \subseteq C_i$ or $C_i \cap C_j = \emptyset$.

Although hierarchical clustering algorithms have many advantages over partitioning clustering algorithms, hierarchical clustering algorithms are typically slower due to their quadratic run time complexity. Hence, hierarchical algorithms do not scale well when compared to partitional clustering algorithms. Specifically, with many real world applications generating very large datasets, analysis of large data using hierarchical clustering becomes difficult and it becomes almost impossible to render the results on a single machine. Also, the fact that hierarchical clustering algorithms require the (dis)similarity measure between all pairs of data objects, introduces a constraint of computing the algorithm on a single machine. This constraint does not allow the algorithm to take advantage of parallel systems easily, making the algorithm essentially sequential.

This dissertation proposes a way to parallelize the hierarchical clustering algorithm HDBSCAN* [7] by data parallelism. HDBSCAN* is an hierarchical clustering algorithm that combines the aspect of density-based clustering, where the data objects in a dense region are separated from objects belonging to other dense regions, and hierarchical clustering, using hierarchical density estimates. HDBSCAN* combines the advantages of both density-based clustering (separating clusters based on their density) and hierarchical clustering (forming a cluster hierarchy instead of partitions at a single density level).

In this dissertation we first present a parallel version to construct the exact HDBSCAN* hierarchy, called *"Random Blocks Approach"*. We evaluate this approach with varying parameter and discuss why the approach might not be scalable with respect to the input parameter. The *"Random Blocks Approach"* is a generalized version based on the parallelization of the Single Linkage algorithm. Then we propose a faster method to construct an approximate version of HDBSCAN* hierarchy using parallel systems, called *"Recursive Sampling*

Approach". The method is based on building the hierarchy based on the sampled data objects, extracting the clusters using the HDBSCAN* hierarchy and recursively refining each cluster to construct an approximate version of the hierarchy. To improve the accuracy, we incorporate a data summarization technique called *Data Bubbles* in to the Recursive Sampling Approach. This technique helps in identifying the structure of the complete dataset using only few summarized structures.

The rest of the dissertation is organized as follows. In Chapter 2, we discuss various Hierarchical Algorithms, followed by an overview of existing Parallel Hierarchical Clustering Algorithms in Chapter 3. Chapter 4 explains the HDBSCAN* algorithm in detail and a method of cluster extraction. Chapter 5 proposes a parallel version of HDBSCAN* called Parallel HDBSCAN* or PHDBSCAN. Chapter 6 introduces a Data Summarization technique that helps in improving the accuracy of PHDBSCAN. Chapter 7 proposes an implementation of the PHDBSCAN algorithm using the MapReduce Framework. Chapter 8 presents an extensive experimental evaluation of the proposed methods.

Chapter 2

Hierarchical Clustering Algorithms

Hierarchical clustering [6] is a method of cluster analysis that builds a hierarchy of clusters, a structure that is more informative than the unstructured set of clusters returned by flat clustering. A hierarchical clustering hierarchy is built gradually where objects change their cluster membership iteratively between different sub-clusters at different levels. Hierarchical clustering algorithms differ from *Partitioning Clustering* algorithms, which generates various partitions and evaluates them based on some criterion. In an hierarchical approach, the clusters can be obtained at different levels of granularity. Hierarchical clustering methods can be further categorized into two broad categories, *agglomerative* (bottom-up) and *divisive* (top-down) approaches. The process of agglomerative clustering is initiated by considering every data object in the dataset to be a singleton cluster. It then recursively merges two or more clusters into one cluster, based on an appropriate criteria or cost function, until all objects belong to one cluster. A cost function is a measure of similarity or dissimilarity between any two objects, and is usually defined by

$$CostFunction,\ \theta = d_{sim}(\cdot, \cdot)$$

This is opposed to the divisive approach where all the data objects are initially considered to be one large cluster and gradually each cluster splits into two or more clusters based on some criteria.

Hierarchical clustering has several advantages over partitioning clustering approach. Hierarchical approaches have the flexibility of forming clusters at different levels of granularity. Once the complete hierarchy is created, clusters can be extracted in a simple way by making a horizontal cut at any given level based on the number of clusters required by the application or the level at which clusters are to be formed. The clusters and sub-clusters are represented in the form of a tree-like structure called *"dendrogram"*.

Dendrogram: A Dendrogram is an important tool for visualizing a hierarchy.

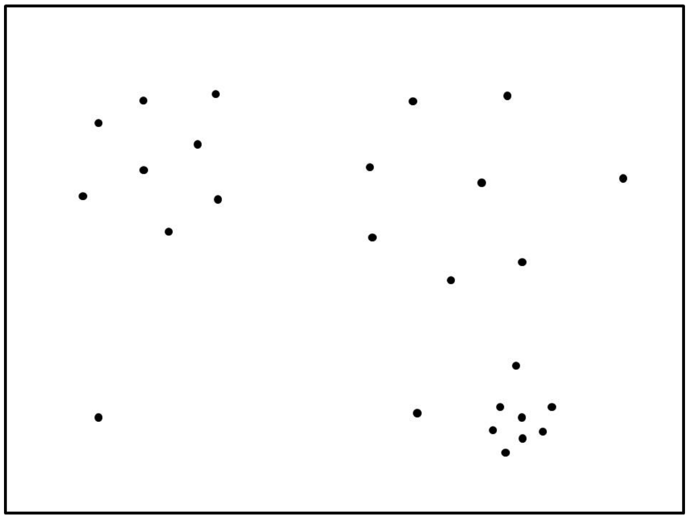

Figure 2.1: Sample Data

A dendrogram is a tree like diagram that records the sequences of clusters that are formed as a result of merging or splitting of sub-clusters. It represents the similarity among objects or group of objects. A dendrogram gives a complete picture of the hierarchical clustering on a given dataset.

A dendrogram consists of *leaves* and *clades*. The terminal nodes are called *leaves* and *clades* represents branches from where the (sub)cluster is split (divisive clustering approach) or merged (agglomerative approach). The height (interchangeably used as distance in this context), as shown in figure 2.2, is a measure of similarity or dissimilarity that exists between the clusters that are merged or split. The arrangement of clades represents how similar the data objects are, with respect to other clades and leaves. Dendrograms also form a powerful tool to visualize and interpret the outliers. Figure 2.2 shows

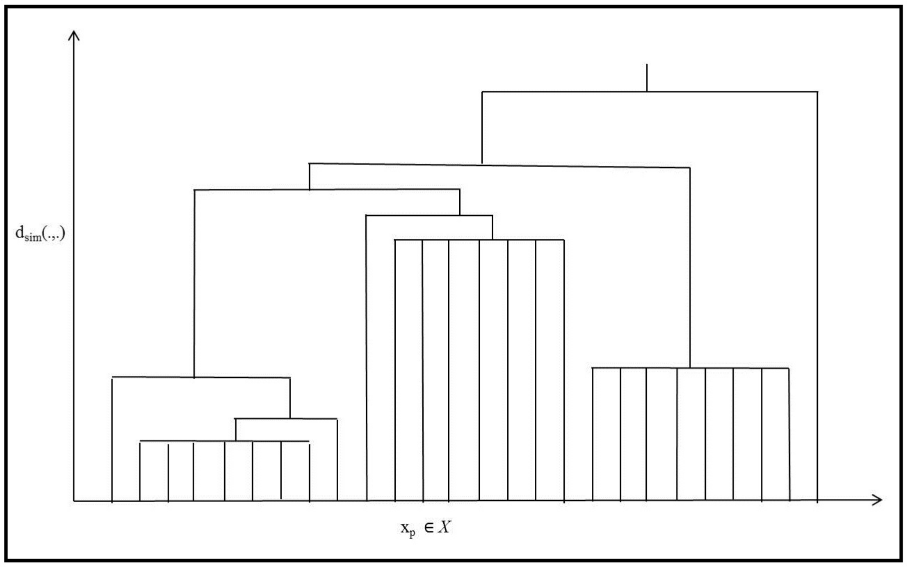

Figure 2.2: Sample Dendrogram for the Data in figure 2.1

a dendrogram drawn corresponding to the data in figure 2.1.

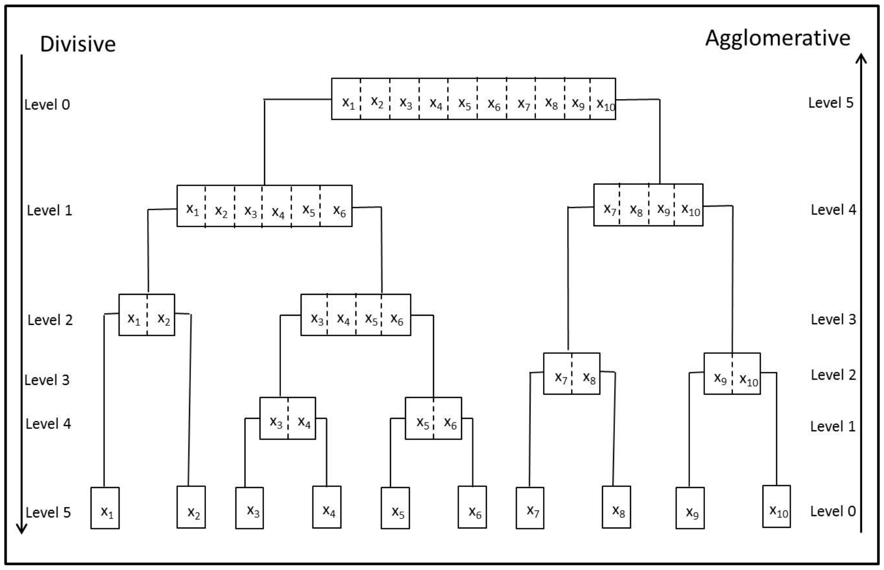

Figure 2.3: Agglomerative vs. Divisive Clustering

2.1 Agglomerative Clustering Approach

The agglomerative approach of hierarchical clustering starts in a state where all the data objects form a set of singleton clusters. To merge one cluster with another, a metric is required. This metric could be one of the Linkage Metric to measure the inter-cluster (dis)similarity or a compactness criterion that measures the degree of closeness among the objects within a cluster at any given level of granularity.

2.1.1 Linkage

The correspondence between any hierarchical system of clusters and a particular type of distance measure was proposed in [6]. The algorithm starts by merging the most similar pair of clusters to form a cluster. Different linkage criteria give rise to different hierarchical clustering techniques.

The generic algorithm for the agglomerative linkage is as follows

1. Initially, start with n objects of dataset X as n clusters where $n = \mid X \mid$ as n clusters, $C = \{C_i\}, 1 \leq i \leq n$
2. Until there is only one cluster remaining, i.e., $\mid C \mid = 1$, do the following
    (a) Find the two most similar clusters among the available clusters in C, $min_{similarity}(C_j, C_k)\ \forall\ C_j, C_k \in C$.
    (b) Combine them to form a single cluster $C_{j,k} = C_j \cup C_k$.
    (c) Remove the clusters C_j and C_k from the set C and add the cluster $C_{j,k}$.

The $min_{similarity}(C_j, C_k)$ defines the linkage method for clustering. The following three examples of well-known linkage criteria given by

Single Linkage The single linkage is given as

$$d_{single} = min_{similarity}(C_j, C_k) = {}_{x_j \in C_j, x_k \in C_k}^{min} \Big\{ d(x_j, x_k) \Big\}$$

where the sets C_j, C_k contains all the data objects in their respective (sub)clusters and d is the distance measure.

Complete Linkage The complete linkage is given by

$$d_{complete} = min_{similarity}(C_j, C_k) = \overset{max}{x_j \in C_j, x_k \in C_k}\Big\{d(x_j, x_k)\Big\}$$

Average Linkage The similarity measure for average linkage is given by

$$d_{avg} = min_{similarity}(C_j, C_k) = \frac{1}{n_i.n_j} \sum_{x_j \in C_j} \sum_{x_k \in C_k} d(x_j, x_k)$$

where d_{avg} is the average distance among all pairs of objects between the sets of C_j, C_k.

2.1.2 BIRCH

Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH) [8] was proposed to achieve quick clustering over large datasets. The algorithm was aimed at reducing the overall computational complexity of clustering, thus allowing hierarchical algorithms to be scalable to large databases and reduce the number of I/O operations. BIRCH finds the clusters in two phases, by building clusters incrementally and dynamically. In the first phase, also called *pre-clustering phase*, compact summaries are created for dense regions. The compact summaries are represented in the form of a *"Cluster Feature Tree"* (CF-Tree) using *"Cluster Feature vectors"* (CF vectors). Clustering Feature vector is a triple given by the number of points, the linear sum of features and sum of squares of features across every dimension.

$$ClusterFeature,\ \vec{CF} = (n, \vec{LS}, SS)$$

where n =number of points

$$LS = \sum_{i=1}^{N} \vec{x_i}$$
$$SS = \sum_{i=1}^{N} \vec{x_i}^2$$

The CF-Tree contains groups of data set into sub-clusters which are compact and approximate. A CF-tree is a height-balanced tree with two parameters, branching factor B and threshold T. Each non-leaf node consists of at most B entries and each leaf node must have its diameter (or radius) less than the threshold T. The CF-Tree is built incrementally where every data object is added to the tree by traversing it and adding it to the closest found leaf node. If the leaf node can absorb this new data object such that the leaf node does not violate the threshold requirement of T, the sufficient statistics of the leaf node is updated. Else the leaf node is split into two (explained in detail in [8]). Each leaf node represents a cluster made up of all the sub-clusters represented by its entries.

The second phase of BIRCH algorithm scans all leaf nodes in the initial CF-tree to rebuild a smaller CF-tree, while removing outliers and grouping dense sub-clusters into a larger one. The next step is to cluster the summaries of all leaf nodes using an existing clustering algorithm. BIRCH scales linearly with respect to the number of objects in the dataset and the clusters can be formed in a single scan. BIRCH can handle only numeric data and is sensitive to the order of the input of the data objects (splitting of leaf nodes are based on the order of the inserted data objects). It works well when there are spherical clusters because it uses the notion of diameter to control the boundary of a cluster.

2.1.3 CURE

Clustering Using REpresentatives (CURE) [9] is an algorithm developed to overcome some of the shortcomings of BIRCH. The CURE algorithm uses several representative points for clustering. These representative points can be more than one per cluster. This allows the algorithm to represent arbitrary shaped clusters. The algorithm starts with a constant number of points called *"scattered points"*. These scattered points are selected randomly by sampling the dataset of a substantial size that will preserve information about the geometry of clusters. The first phase of CURE partitions a randomly drawn sample into p partitions. Each partition approximately contains $\frac{n}{p}$ data objects, where n is the input. Each partition is partially clustered until the final number of clusters in each partition reduces to $\frac{n}{pq}$, for some constant $q > 1$. Then a second clustering pass is run on the $\frac{n}{q}$ partial clusters for all partitions to get the final clustering result.

2.1.4 ROCK

ROCK (RObust Clustering using linKs) [10] is an agglomerative hierarchical algorithm for categorical attributes where the clustering is based on *"links"* between data objects instead of distances. The algorithm defines an object x_j as a *neighbor* of an object x_k if $d_{sim}(x_j, x_k) \geq \theta$ for some threshold θ, given a similarity function $d_{sim}(\cdot, \cdot)$. The number of links between data objects x_j and x_k, given by $link(x_j, x_k)$, is the number of common neighbors between x_j and x_k. If $link(x_j, x_k)$ is large, then the probability of x_j and x_k belonging to the same cluster is assumed to be high. Clusters are defined as a set of objects with a high degree of connectivity; pairs of objects inside a cluster have a high number of links, on an average. ROCK uses a *"goodness measure"* which measures the *goodness* of clusters. The *goodness measure* for merging clusters C_j and C_k is given by

$$g(C_j, C_k) = \frac{link(C_j, C_k)}{(n_j + n_k)^{1+2f(\theta)} - (n_j)^{1+2f(\theta)} - (n_k)^{1+2f(\theta)}}$$

where $n_j = \mid C_j \mid$, $n_k = \mid C_k \mid$ and $n_i^{1+2f(\theta)}$ gives an estimate of the total number of links in cluster C_i.

The pair of clusters for which the goodness measure is maximum is the best pair of clusters to be merged. Although it is intuitive to merge pairs of clusters with a large number of cross-links (links between clusters), the goodness measure is computed by normalizing the number of cross-links with an estimate of the total number of links in the clusters. This is because a large cluster typically would have a larger number of cross-links with other clusters and the larger cluster may be merged with smaller clusters or outliers.

2.1.5 CHAMELEON

The hierarchical agglomerative algorithm CHAMELEON [11] uses a dynamic modelling approach in cluster aggregation. It uses the connectivity graph G

corresponding to the k-Nearest Neighbor model sparsification of the connectivity matrix. That is, the edges from a given object to their k-Nearest Neighbor are preserved and rest of the edges are pruned. Chameleon operates on the sparse graph in which nodes represent the data objects and weighted edges represent the similarities among the data objects. Data objects that are far apart are completely disconnected. Sparsity of the graph leads to computationally efficient algorithms.

CHAMELEON is a two-phase algorithm. The first phase uses a graph partitioning algorithm to cluster the data items into several relatively smaller sub-clusters. During the second phase, it uses an algorithm to find the clusters by repeatedly combining these sub-clusters in agglomerative fashion. It uses a dynamic modelling framework to determine the similarity between pairs of clusters by considering the following two measures

1. Relative Interconnectivity, given by

$$RI(C_j, C_k) = \frac{\mid EC(C_j, C_k) \mid}{\frac{|EC(C_j)|+|EC(C_k)|}{2}}$$

   where $EC(C_j, C_k)$, is the absolute interconnectivity given by the edge cut (sum of the weight of edges that connect the two clusters); and $E(C_j)$ and $E(C_k)$ is the internal interconnectivity, given by the sum of edges crossing a min-cut bisection that splits the clusters into two roughly equal parts.

2. Relative Closeness, computed by the average weight of edges connecting the objects in cluster C_j to the objects in cluster C_k, is given by

$$RC(C_j, C_k) = \frac{\overline{SEC}(C_j, C_k)}{\frac{|C_j|}{|C_j|+|C_k|}\overline{SEC}(C_j) + \frac{|C_k|}{|C_j|+|C_k|}\overline{SEC}(C_k)}$$

   where $\overline{SEC}(C_j)$ and $\overline{SEC}(C_k)$ are the average weights of the edges that belong to the min-cut bisector of clusters C_j and C_k, and $\overline{SEC}(C_j, C_k)$ is the average weight of the edges that connect the vertices in C_j and C_k.

2.2 Divisive Clustering Approach

2.2.1 DIANA

DIvisive ANAlysis (DIANA) is one of the few divisive hierarchical algorithms. The DIANA algorithm constructs a hierarchy of clusters, starting with one cluster containing all n objects. The algorithm then divides each cluster until each cluster contains only one data object. At each stage of division, the cluster with the largest diameter is selected. The diameter of a cluster is the largest distance between two data objects within that cluster. To divide the selected cluster, one of the most disparate data object, x_d, with the maximum average dissimilarity to the other data objects in a cluster C is selected. This object x_d initiates a group called the *"splinter"* group. For each data object, $x_p \in C$, the average dissimilarity between x_p and the set of elements that are not in the splinter group as well as the dissimilarity between x_p and the splinter group is computed; an object x_p is then assigned to the group with smaller dissimilarity measure.

$$dissim_{avg}(x_p, C_j) = \sum_{x_j \notin C_s,\ x_j \in C_j} \frac{d(x_p, x_j)}{\mid C_j \mid}$$

$$dissim_{avg}(x_p, C_s) = \sum_{x_j \in C_s} \frac{d(x_p, x_j)}{\mid C_s \mid}$$

where x_p is the disparate data object with maximum dissimilarity $\left(\overset{max}{x_i \in C_j} dissim(x_i, C_j)\right)$

C_s is the splinter group

C_j is the old group from which x_p was selected

2.2.2 MONA

MONothetic Analysis [12] is another divisive hierarchical clustering algorithm which operates on a data matrix of binary variables. A variable is a feature associated with a data object which can have only two values, 0 and 1. Each division is performed using a well-selected single variable and hence the name *"Monothetic"*. The algorithm starts with a single variable, and then divides the whole dataset into two groups, with values 0 and 1. The variable that divides the dataset into groups is the variable with the largest association measure among all the available variables.

$$A(V_s) = max(A(V_i));\ V_i \in V$$

where V_s is the selected variable, $A(V_s)$ is the association of V_s and set V contains all the binary variables.

Chapter 3

Parallel Hierarchical Clustering Algorithms

Parallel clustering algorithms have been studied widely, but relatively less research has been done on parallel hierarchical clustering algorithms. This is due to the nature of hierarchical clustering where the hierarchy is constructed based on a (dis)similarity measure computed between all pairs of data objects.

Single LINKage (SLINK) clustering algorithm [13] is one the most studied hierarchical clustering algorithms for parallelization [14] [15] [16] [17]. The first implementation of a parallel version of the SLINK algorithm was discussed in [18], which proposed an implementation on a Single Instruction Multiple Data (SIMD) array processor. SIMD is a class of parallel computers which has multiple processing elements that can perform the same operation simultaneously on multiple data objects. These processors can execute a single instruction across all the processing elements at any given time. In [19], the parallelization of Single Linkage was translated to the problem of parallelizing a Minimum Spanning Tree (MST) with data objects as vertices of a graph and the distance between the objects as edge weights of edges connecting the vertices. The algorithm generates an MST in parallel, using $\frac{N}{lgN}$ processors in a time proportional to $\mathcal{O}(NlgN)$. The idea is to run the algorithm on a *tree structured searching machine* [19] [20], containing different types of nodes with different functionalities.

Another work on parallel hierarchical clustering algorithms is presented in [14].

The author proposed two algorithms based on the memory architecture of the system used for parallelization. The first algorithm finds a Single Linkage hierarchy in parallel on a shared memory architecture. The metric is calculated on an n processor PRAM (Parallel Random Access Machines), where n is the number of data objects that allow the processors to access a single parallel memory simultaneously. Each cluster will be the responsibility of one processor. When two clusters are agglomerated, the lower numbered processor corresponding to the two clusters takes over full responsibility of the new cluster. If the other processor is no longer responsible for any cluster, it remains idle. The algorithm can be described as follows

1. Create a two dimensional array to store the inter-cluster distance between all data objects and a one-dimensional array storing the nearest neighbor (and its corresponding distance) of each cluster.
2. Identify the pair of clusters separated by the smallest distance between them and agglomerate them.
3. Send a broadcast message about the two clusters that were merged.
4. Update the data structure where each processor updates the inter-cluster distance array to reflect the new distance between its clusters and the new merged cluster. Each processor updates a single location in the nearest neighbor array and the corresponding column in the inter-cluster distance matrix.
5. Repeat the steps until there is only one cluster.

Similar approaches for the average link and complete link metrics on a PRAM are also mentioned in [14]. The second algorithm for parallelizing Single Linkage is designed for a distributed memory architecture, based on the parallel minimum spanning tree algorithm given in [21]. The algorithm is described as follows

1. Using some data structure D, keep track of the distances between each object and the current minimum spanning tree, say $MST(X')$ for a

dataset X.

2. Find the data object $x_p \notin MST(X')$ and $x_p \in X$ closest to the MST.

3. Add x_p to $MST(X')$, $MST(X') = MST(X') \cup x_p$.

4. Update the data structure D.

5. Repeat the steps until all the data objects are added to $MST(X')$.

The notion of parallelization comes into picture when each processor responsible for a data object not in the minimum spanning tree identifies the distance to the minimum spanning tree. The data structure D is distributed across the processors such that the processor responsible for the cluster stores the distances for that cluster. Each processor identifies the nearest distance by comparing the current distance with newly added object to the tree and updating it with the minimum distance.

Similar approaches have been examined across different architectures such as GPUs [22] by parallelizing the computation of the minimum distance between clusters. An OpenMP implementation for distributed memory architectures was provided in [23] similar to that provided by [14] with a message passing across multiple systems instead of processing elements. This improves the overall running time from $\mathcal{O}(n^2)$ to $\mathcal{O}(\frac{n^2}{p})$ where p is the number of available distributed systems.

Some of the recently developed approaches for parallel computation of the Single Linkage hierarchy are described in the following sections.

3.1 PARABLE

PArallel, RAndom-partition Based hierarchicaL clustEring (PARABLE) [24] was designed for the MapReduce Framework. It is a two step algorithm, with the first step computing local hierarchical clusterings on distributed nodes and the second step integrating the results by a proposed *dendrogram alignment technique*. First, the dataset is divided into random partitions. Given a dataset

X, it is divided into k partitions. The sequential algorithm is then computed on these partitions locally to form intermediate dendrograms. These intermediate dendrograms are then aligned with each other to form the final dendrogram by recursively aligning the nodes from root to the leaves, by comparing the nodes of two given dendrograms using a similarity measure. The similarity measure is given by

$$Similarity(C_j, C_k) = \frac{(LS_j \times LS_k + SS_j + SS_k)}{\mid C_j \mid \times \mid C_k \mid}$$

where LS_j is the linear sum of all objects in cluster C_j

SS_j is the sum of squares of all objects in cluster C_j

$\mid C_j \mid$ is the cardinality of the cluster C_j

The authors claim that the "alignment procedure is reasonable when the dendrograms to be aligned have similar structure" [24] and that a good sampling strategy can lead to local dendrograms being similar. During the dendrogram alignment technique, the algorithm fails to address the fact that addition of new branches in a dendrogram (addition of new data objects to the dataset) also affect the height at which the clades are formed.

3.2 CLUMP

CLUstering through MST in Parallel (CLUMP) [25] is one of first algorithms that addresses the parallelization of hierarchical algorithms based on overlapping datasets that can be processed on distributed nodes without any communication between them. The authors use the *"Linear Representation"*(LR) of the construction of an MST. LR is a list of elements of dataset X whose sequential order is the same as the order in which these elements got selected by the MST construction algorithm (which is *Prims algorithm* for constructing an MST). CLUMP operates on a complete graph $G_X = (V_X, E_X)$ of a dataset X in the following way

1. Partition the dataset G_X into k subgraphs, $\{G_i = (V_i, E_i)\}, 1 \leq i \leq k$ where V_i is a set of vertices in G_i, $V_j \subset V_X$ and E_i is a set of edges in G_i, $E_j \subset E_X$.

2. Define Bipartite Graphs $B_{ij} = \{V_i \cup V_j, E_{ij}\}$ where $E_{ij} \subset E_X$ is a set of edges between vertices in V_i and V_j such that $i \neq j$.

3. Construct an MST T_{ii} on each G_i, and MST T_{ij} on each B_{ij} in parallel.

4. Build a new graph $G^0 = \bigcup T_{ij}, 1 \leq i \leq j \leq k$.

5. Construct an MST on this graph G^0.

This algorithm has been adopted by many other approaches to find the Single Linkage hierarchy in parallel. This approach is also the basis of one our approaches explained later in section 5.1.

3.3 SHRINK

SHaRed-memory SLINK (SHRINK) [26] is a scalable algorithm to parallelize Single Linkage hierarchical clustering algorithm. The strategy is to partition the original dataset into overlapping subsets of data objects and compute the hierarchy for each subset using Single Linkage. Construct the overall dendrogram by combining the solutions for the different subsets of the data. The algorithm for SHRINK is described as follows:

1. Partition the original dataset X into k subsets of approximately the same size, $X'_1, X'_2, ..., X'_k$.

2. For all pairs of subsets $(X_i, X_j); 1 \leq i, j \leq k, i \neq j$, compute the dendrogram of the dataset $X_i \cup X_j$.

3. Combine the dendrograms, two at a time, by processing the cluster merges in order of increasing height, eliminating merges that combine data objects already in the combined dendrogram. Repeat this until only one dendrogram exists.

The authors have proposed the Union-Find (Disjoint Set) data struucture to maintain the cluster membership for each data object as the dendrograms are combined.

Other algorithms such as PINK [27] and DiSC [1] are based on the same idea as CLUMP and SHRINK. While PINK is designed for a distributed memory architecture, DiSC is implemented using the MapReduce Framework.

3.4 DiSC

Distributed Single Linkage Hierarchical Clustering (DISC) [1] algorithm is based on the idea of SHRINK (explained in section 3.3) and implemented using the MapReduce framework. The data is divided into overlapped subsets of data which are processed individually at different distributed nodes to form the sub-solutions. The sub-solutions are then combined to form an overall solution.

The MapReduce implementation of DiSC consists of two rounds of MapReduce jobs (MapReduce framework is explained in detail in Chapter 7). The first round of MapReduce, consists of a Prim's Mapper, which builds a minimum spanning tree on the subset of data sent to it and a Kruskal Reducer, which uses a K-way merge to combine and get the intermediate result. The Kruskal Reducer uses a UnionFind data structure [27] to keep track of the membership of all the connected components and filter out any cycles. The second MapReduce job, called the Kruskal-MR job consists of a Kruskal Mapper which is just an identity mapper (rewrites the input as output) and a Kruskal Reducer, which essentially does the same work as the reducer of first round. The flow of DiSC algorithm on MapReduce is illustrated in figure 3.1.

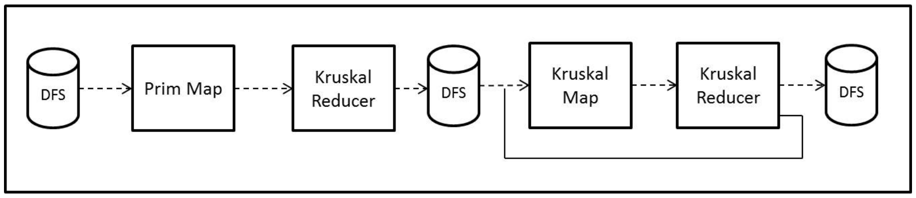

Figure 3.1: DiSC algorithm on MapReduce (adapted from [1])

Chapter 4

Hierarchical DBSCAN*

Hierarchical DBSCAN* or HDBSCAN* [7] was a conceptual and algorithmic improvement over the OPTICS [28] algorithm with a single input parameter called the m_{pts}, a classic smoothing factor for density estimates. The hierarchy produced by HDBSCAN* can be used as a base for various other post-processing tasks. One of these tasks include creating a compact hierarchy, a hierarchy which only consists of those density levels where there is prominent change, or extracting a set of the most prominent non-overlapping clusters without using a single density threshold by making local cuts through the cluster tree at different density levels. The hierarchy can also be transformed for visualization. Some of the ways to visualize the hierarchy are by converting the hierarchy into a reachability plot [28], a silhouette-like plot [29], a dendrogram or a compact cluster tree. The authors in [7] have also explained methods to identify outliers and measuring the level of outlierness by providing a score, called *"Global-Local Outlier Score from Hierarchies"(GLOSH).*

The HDBSCAN* algorithm is a hierarchical version of DBSCAN* which is a reformulation of DBSCAN [30]. DBSCAN* conceptually finds clusters as connected components of a graph in which the objects of dataset X are vertices and every pair of vertices is adjacent only if the corresponding objects are ϵ-reachable with respect to the parameters ϵ and m_{pts}. The parameter m_{pts} is a smoothing factor which has been well studied in literature [28] [31] [32]. Different density levels in the resulting density-based cluster hierarchy will then correspond to different values of the radius ϵ.

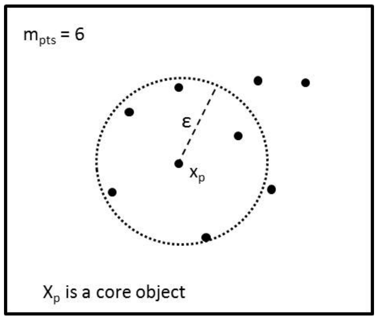

Figure 4.1: Core Object

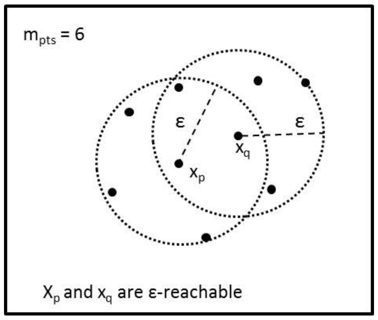

Figure 4.2: $\epsilon - reachable$

In the following section, we describe the algorithm HDBSCAN* in detail as presented in [7].

The following definitions help to formulate a relationship between DBSCAN* with its hierarchical version HDBSCAN*.

4.1 Algorithm DBSCAN*

Let $X = \{x_1,x_2,...,x_n\}$ be a set of n d-dimensional data objects. Their pairwise distances can be represented by a $[n \times n]$ matrix, X_{pd}, with any element of the matrix $X_{i,j} = d(x_i, x_j)$, $1 \leq i, j \leq n$, where $d(x_i, x_j)$ is some distance measure between data objects x_i and x_j.

Definition 1. Core Object: An object x_p is called a core object w.r.t. ϵ and m_{pts} if its ϵ-neighborhood, $N_\epsilon(\cdot)$, contains at least m_{pts} objects, i.e., $| N_\epsilon(x_p) | \geq m_{pts}$, where $N_\epsilon(x_p) = \{x \in X \mid d(x, x_p) \leq \epsilon\}$ and $| \cdot |$ denotes cardinality of the enclosed set. An object is called *"noise"* if the object is not a core object.

An example of a core object is shown in figure 4.1 with x_p being the core object w.r.t parameters ϵ and $m_{pts} = 6$.

Definition 2. ϵ-reachable: Two core objects x_p and x_q are ϵ-reachable w.r.t. ϵ and m_{pts} if $x_p \in N_\epsilon(x_q)$ and $x_q \in N_\epsilon(x_p)$.

An example of ϵ-reachable objects x_p and x_q is shown in figure 4.2 where both

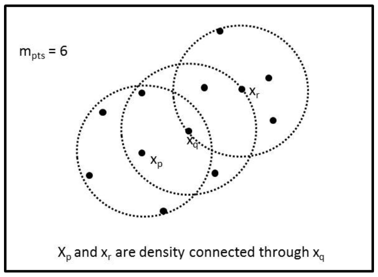

Figure 4.3: Density-Connected Objects

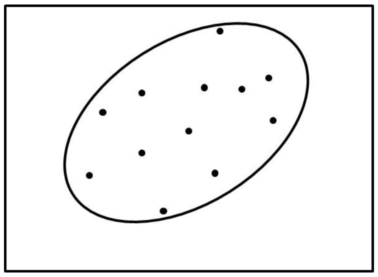

Figure 4.4: Example of a Cluster

the objects, x_p and x_q are within the ϵ-neighborhood of each other.

Definition 3. Density-Connected: Two core objects x_p and x_q are density connected w.r.t. ϵ and m_{pts} if they are directly or transitively ϵ-reachable.

Figure 4.3 explains the concept of Density-Connected objects where x_p, x_q and x_r are density-connected (x_p and x_r are transitively ϵ-reachable through x_q).

Definition 4. Cluster: A cluster C w.r.t. ϵ and m_{pts} is a non-empty maximal subset of X such that every pair of objects in C is density-connected. An example of a cluster is shown in figure 4.4.

4.2 HDBSCAN*

HDBSCAN* is based on the concept that a hierarchy can be built from different levels of density based on different values of ϵ. HDBSCAN* can be explained based on the following definitions:

Defintion 5. Core Distance: The core distance, $d_{core}(x_p)$, of an object $x_p \in X$ w.r.t. m_{pts} is the distance between x_p to its m_{pts}-nearest neighbor (including x_p).

The core distance is defined as the minimum radius ϵ such that a data object x_p satisfies the condition for it to be a core object with respect to ϵ and m_{pts}.

Definition 6. ϵ-Core Object: An object $x_p \in X$ is called an ϵ-core object for every value of ϵ that is greater than or equal to the core distance of x_p

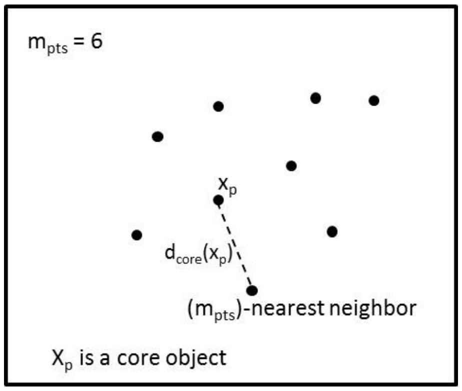

Figure 4.5: Core Distance of an Object

w.r.t. m_{pts}, i.e., if $d_{core}(x_p) \leq \epsilon$.

Definition 7. Mutual Reachability Distance: The mutual reachability distance between two objects x_p and x_q in the dataset X w.r.t. to m_{pts} is defined as

$$d_{m_{reach}}(x_p, x_q) = max\big(d_{core}(x_p), d_{core}(x_q), d(x_p, x_q)\big)$$

Definition 8. Mutual Reachability Graph: A Mutual Reachability Graph can be defined as a complete graph, $G_{m_{reach}, m_{pts}}$ or $G_{m_{reach}}$, w.r.t m_{pts}, in which the data objects of dataset X are the vertices, V, and edges, E, that exist between every pair of vertices. The weights of edges between any two vertices is defined by the mutual reachability distance of the corresponding data objects in X.

$$G_{m_{reach}, m_{pts}} = G_{m_{reach}} = (V, E)$$

$$\begin{aligned} \text{where} \quad & V = \{v_i\}; \quad 1 \leq i \leq n \\ & E = \{v_i, v_j, d_{m_{reach}}(x_i, x_j)\}; \quad 1 \leq i, j \leq n \end{aligned}$$

An example of a Minimum Spanning Tree (MST) generated from a complete graph is shown in figure 4.6 where edges that form the MST are shown using solid lines and edges that are a part of the complete graph but not in the MST are shown using the dotted line.

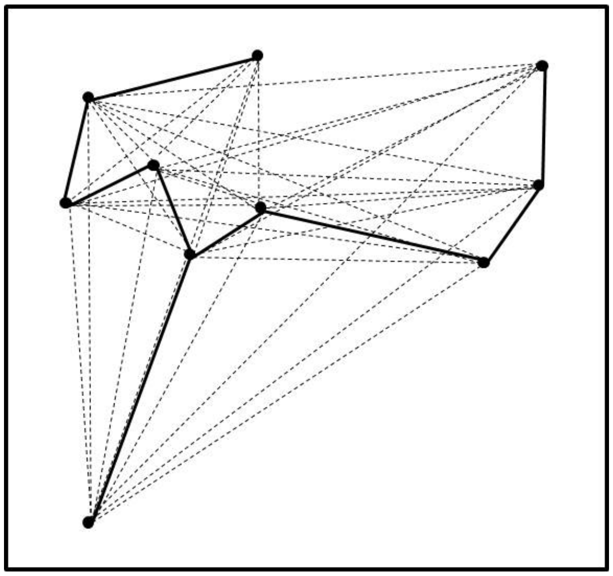

Figure 4.6: Minimum Spanning Tree from a Complete Graph

From defintions 4, 6 and 8, it can be deduced that the clusters created according to DBSCAN* w.r.t partitions for $\epsilon \in [0, \infty)$ can be produced in a nested and hierarchical way by removing edges in decreasing order of weight from the graph $G_{m_{reach}}$. It is equivalent to applying Single Linkage on a transformed space of mutual reachability distances and then making a horizontal cut at ϵ. The set of connected components obtained are *"clusters"* while the singleton objects are *"noise"* objects. All possible levels in a hierarchy can be extracted by removing one edge at a time with decreasing values of ϵ starting with the maximum value of ϵ from the graph $G_{m_{reach}}$.

A density-based cluster hierarchy represents the fact that an object o is noise below the level l that corresponds to o's core distance. To represent this in a dendrogram, we can include an additional dendrogram node for o at level l representing the cluster containing o at that level or higher. To construct such a hierarchy, an extension of Minimum Spanning Tree (MST) of the Mutual Reachability Graph $G_{m_{reach}}$ is required. The MST is extended by adding *"self-edges"* to each vertex with an edge weight equal to that of the core distance of o, $d_{core}(o)$. This extended MST can be used to construct the extended dendrogram by removing edges in decreasing order of weights.

The hierarchy can be computed by constructing an MST on the transformed space of mutual reachability distances. The MST can be constructed quickly by removing edges in decreasing order of the edge weights [4].

The pseudocode for HDBSCAN* algorithm is shown in Algorithm 1.

Algorithm 1: HDBSCAN* Main Steps

```
Input: Dataset $X$, Parameter $m_{pts}$
Output: HDBSCAN* hierarchy
  1. Given a dataset $X$, compute the core distances of all the data objects
     in $X$.
  2. Compute a Minimum Spanning Tree of the Mutual Reachability
     Graph, $G_{m_{reach}}$.
  3. Extend the MST to obtain $MST_{ext}$, by adding a "self loop edge" for
     each vertex with weight equal to that of its core distance, $d_{core}(x_p)$.
  4. Extract the HDBSCAN* hierarchy as a dendrogram from $MST_{ext}$.
     (a) All the objects are assigned to the same label, thus forming the
         root of the tree.
     (b) Iteratively remove all edges from $MST_{ext}$ in decreasing order of
         weights.
          i. Edges with the same weight are removed simultaneously.
         ii. After removal of an edge, labels are assigned to the connected
             components that contain one vertex of the removed edge. A
             new cluster label is assigned if the component has at least one
             edge in it, else the objects are assigned a null label, indicating
             it to be a noise object.
```

4.3 Hierarchy Simplification

A method of hierarchy simplification has also been proposed in [7]. The simplification of HDBSCAN* hierarchy is based on an observation about estimates of the level sets of continuous-valued probability density function (p.d.f.), which refers back to Hartigan's concept of *"rigid clusters"*. In a divisive approach, for a given p.d.f., there are only three possibilities for the evolution of the connected components of a continuous density level sets when increasing the density level.

1. The component shrinks but remains connected, up to a density threshold, at which either
2. The component is divided into smaller ones, called a *"true split"*, (or)
3. The component disappears.

Based on the conditions given above, the HDBSCAN* hierarchy can be simplified into a hierarchy which only consists of those levels where there is a *"true split"* or the levels at which an existing cluster disappears. Other levels of the hierarchy, i.e., values of ϵ at which the data objects are assigned null labels making them *"noise"* objects are the levels where a particular component has shrunk, and those levels are not explicitly maintained in a simplified hierarchy.

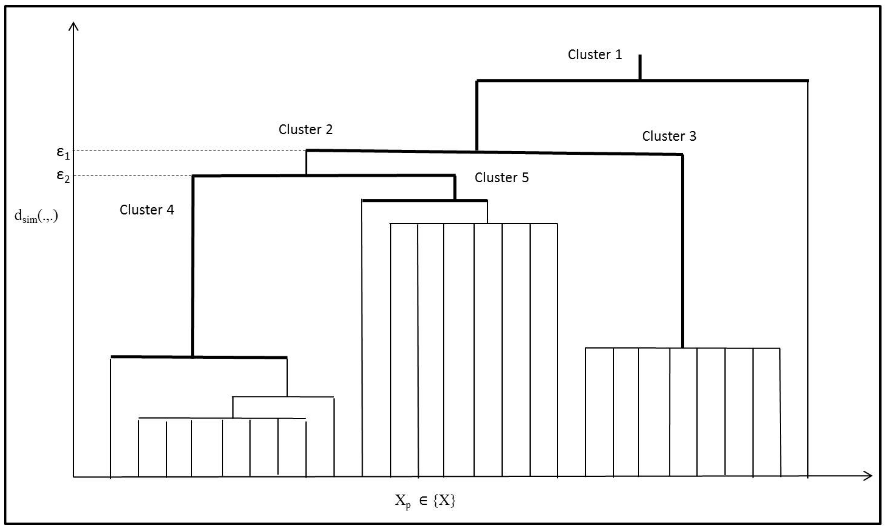

Figure 4.7: Cluster Dendrogram for a dataset

Using optional parameter m_{clSize}: Usually many applications require that a connected component, in order to be considered a cluster, need at least a few data objects that are very similar to each other. This requirement can easily be incorporated in the HDBSCAN* hierarchy by the use of a parameter m_{clSize} which indicates the minimum size of a connected component to be considered

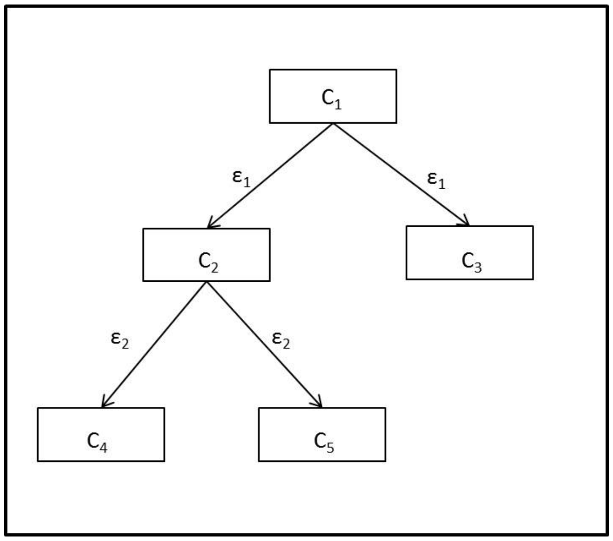

Figure 4.8: Cluster Tree for the given Dendrogram in 4.7

a cluster. Therefore the step 4.(b).(ii) of Algorithm 1, can be generalized to accommodate the parameter m_{clSize} and is described in Algorithm 2.

Algorithm 2: Step 4.(b).(ii) of Algorithm 1 with optional parameter $m_{clSize} \geq 1$

4.(b).ii. After removal of each edge, process the cluster that contained the removed edge (one at a time) as follows:

- Label spurious sub-components as noise by assigning them the null label. If all the sub-components of a cluster are spurious, then the cluster has disappeared. A sub-component is termed as spurious, if the number of vertices in the sub-component are less than m_{clSize}.
- If there is a single sub-components of a cluster that is not spurious, then the original cluster label is maintained. This means that the cluster has shrunk.
- If there are two or more sub-component of a cluster that are not spurious, assign new labels to each of them. This means that the parent cluster has not split into two clusters.

4.4 Computational Complexity of HDBSCAN*

There are two scenarios over which the computational complexity has been calculated

4.4.1 Scenario 1: Dataset X is available

Given a dissimilarity function $d(\cdot,\cdot)$ that is used to measure the distance between two data objects, the distance between each pair of objects can be computed in $\mathcal{O}(d)$ time, where d is the dimensionality of any given data object. Therefore, step 1 of Algorithm 1 can be computed in $\mathcal{O}(dn^2)$ time. The time taken by Step 2 of the algorithm to construct an MST is $\mathcal{O}(n^2+e)$, assuming a list based search version of Prim's Algorithm, where e is the number of edges in $G_{m_{reach}}$. Since it is a complete graph, the number of edges, $e = \frac{n(n-1)}{2}$. This allows the MST to be built in $\mathcal{O}(n^2)$ time. Step 3 can be computed in $\mathcal{O}(n)$ time, since the distances have already been computed and this step requires an addition of n self-edges to the graph. Step 4 requires sorting of edges, which can be computed in $\mathcal{O}(nlogn)$. Step 4 relabels according to Algorithm 2 by removing edges at different values of ϵ (in decreasing order of ϵ), which can be achieved in $\mathcal{O}(n^2)$. Therefore, the overall algorithm has a computational time complexity of $\mathcal{O}(dn^2)$.

The overall space taken by this scenario is only the space to store the dataset X which is $\mathcal{O}(dn)$ for n d-dimensional data objects since the matrix X_{pd} is computed on demand.

4.4.2 Scenario 2: Pairwise Distance Matrix, X_{pd} is already given

The availability of the pairwise distance matrix, X_{pd} reduces the overall complexity of the algorithm to $\mathcal{O}(n^2)$ instead of $\mathcal{O}(dn^2)$ since there is no requirement for the computation of the distance $d(\cdot,\cdot)$ between any pair of objects and they can be accessed in constant time.

The overall space complexity in this scenario however increases from $\mathcal{O}(dn)$ to $\mathcal{O}(n^2)$, since the matrix X_{pd}, which contains $\mathcal{O}(n^2)$ values is to be stored in the main memory.

4.5 Extraction of Prominent Clusters

A hierarchy is a good representation of the overall structure of the data. Many practical applications however may require to extract a set of non-overlapping clusters. Flat partitioning to extract clusters by a horizontal cut at a single density threshold might not capture the notion of *"true clusters"*, which may exist at different density levels. A procedure to extract the most prominent clusters is given in [7]. This procedure aims at maximizing the overall cluster stability from the set of clusters extracted from local cuts in a HDBSCAN* hierarchy.

Hartigan's model [33] shows that the density-contour clusters of a given density $f(x)$ on $\mathbb{R}$ at a given density level λ are the maximal connected subsets of the level set defined as $\{x \mid f(x) \geq \lambda\}$. DBSCAN* estimates density-contour clusters using a density threshold $\lambda = \frac{1}{\epsilon}$ and a non-normalized k-NN estimate, for $k = m_{pts}$ of $f(x)$, given by $\frac{1}{d_{core}(x)}$.

HDBSCAN* produces all possible DBSCAN* solutions w.r.t. a given value of m_{pts} and all thresholds $\lambda = \frac{1}{\epsilon}$ in $[0,\infty)$. By increasing the density threshold λ (decreasing ϵ), the clusters become smaller and are increasingly associated with denser regions. Components which are connected only by longer edges (lower density thresholds) disappear before the clusters that are associated with higher density thresholds. This gives an intuition that more prominent clusters tend to *"survive"* longer after they appear, which is essentially the rationale behind the definition of *"cluster lifetime"* from classic hierarchical cluster analysis [4].

The lifetime of a cluster in a traditional dendrogram is defined as the length of the dendrogram scale along those hierarchical levels for which the cluster exists. In a HDBSCAN* hierarchy, an existing cluster is considered to be dissolved when the cluster disappears by diminishing into singleton objects, or when the cardinality of the cluster at that level falls below m_{clSize}, or when two or more prominent clusters are formed due to the removal of edges. Since each data object belonging to a cluster can become *noise* at a density different from the density at which the cluster splits or disappears, stability of the cluster not only depends on the lifetime of a cluster in the hierarchy but also on the individual density profiles of all data objects present in that cluster.

The density contributed by a single data object x_p towards a cluster stability is defined as

$$\lambda_{max}(x_p, C_i) - \lambda_{min}(C_i)$$

where, $\lambda_{max}(x_p, C_i)$ is the maximum density at which the data object x_p belonged to the cluster, i.e., the density at which the object x_p or cluster C_i disappears or the density at which the cluster membership of the object is changed (cluster has been split into two or more clusters where the data object now belongs to another cluster); $\lambda_{min}(C_i)$ is the threshold at which the cluster first appears.

Using this definition, the stability $S(C_i)$ of a cluster C_i can be defined as

$$\begin{aligned} S(C_i) &= \sum_{x_j \in C_i} (\lambda_{max}(x_p, C_i) - \lambda_{min}(C_i)) \\ &= \sum_{x_j \in C_i} \left(\frac{1}{\epsilon_{min}(x_j, C_i)} - \frac{1}{\epsilon_{max}(C_i)} \right) \end{aligned}$$

4.6 Cluster Extraction as an Optimization Problem

Let $\{C_2, ..., C_k\}$ be a collection of all clusters in the simplified cluster hierarchy (tree) generated by HDBSCAN*, except the root C_1, and let $S(C_i)$ denote the stability value of each cluster. The problem of extracting the most prominent clusters by flat non-overlapping partitioning, is converted into an optimization problem with the objective of maximizing the overall aggregated stabilities of extracted clusters by

$$\underset{\delta_2,\ldots,\delta_\kappa}{max} J = \sum_{i=2}^{\kappa} \delta_i S(C_i) \tag{4.1}$$

subject to

$$\begin{cases} \delta_i \in \{0,1\}, & i = 2, ..., \kappa \\ \sum\limits_{j \in I_h} (\delta_j) = 1, & \forall h \in L \end{cases}$$

where δ_i indicates whether cluster C_i is included into the flat solution ($\delta_i = 1$) or not ($\delta_i = 0$), $\mathbf{L} = \{H \mid C_h$ is a leaf cluster$\}$ is the set of indexes of leaf clusters, and $I_h = \{\ j \mid j \neq 1$ and C_j is ascendant of C_h (h included) $\}$ is a set of indexes of all clusters on the path from C_h to the root (excluded). The constraints prevent nested clusters on the same path to be selected.

Algorithm 3: Solution to problem (4.1)

```
1. Initialize $\delta_2 = ... = \delta_\kappa = 1$, and, for all leaf nodes, set $\hat{S}(C_h) = S(C_h)$.
2. Starting from the deepest levels, do bottom-up (except for the root)
   **If** $S(C_i) < \hat{S}(C_{i_l}) + \hat{S}(C_{i_r})$ **then**
      set $\hat{S}(C_i) = \hat{S}(C_{i_l}) + \hat{S}(C_{i_r})$ and set $\delta_i = 0$. **else**
      Set $\hat{S}(C_i) = S(C_i)$ and set $\delta_{(\cdot)} = 0$ for all clusters in $C_i$'s subtree.
   **end**
```

To solve problem (4.1), we process every node except the root, starting from the leaves (bottom-up), deciding at each node C_i whether C_i or the best-so-far selection of clusters in C_i's subtrees should be selected. To be able to make this decision locally at C_i, we propagate and update the total stability $\hat{S}(C_i)$ of clusters selected in the subtree rooted at C_i in the following recursive way:

$$\hat{S}(C_i) = \begin{cases} S(C_i) & if\ C_i\ is\ a\ leaf\ node \\ max\{S(C_i), \hat{S}(C_{i_l}) + \hat{S}(C_{i_r})\} & if\ C_i\ is\ an\ internal\ node \end{cases}$$

where C_{i_l} and C_{i_r}, are the left and right children of C_i (for the sake of simplicity, we discuss the case of binary trees; the generalization to n-ary trees is trivial).

Algorithm 3 gives the pseudocode for finding optimal solution to the problem (4.1). Figure 4.9 illustrates a density function, cluster stabilities and excess of mass. Figure 4.10 illustrates the Cluster Tree corresponding to the density function shown in figure 4.9. Cluster extraction through excess of mass and optimizing the objective of maximizing the aggregated cluster stabilities for Algorithm 3 is explained through figures 4.9 and 4.10.

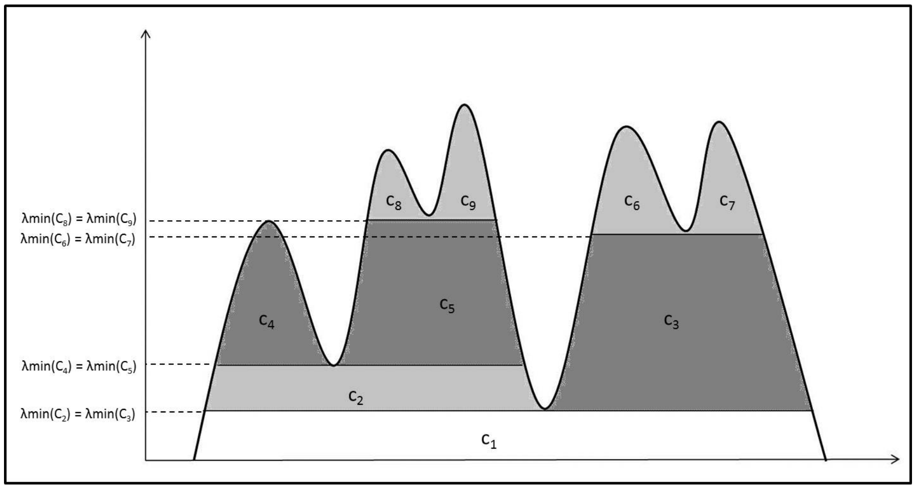

Figure 4.9: Illustration of Density function, Cluster Stability and Excess of Mass

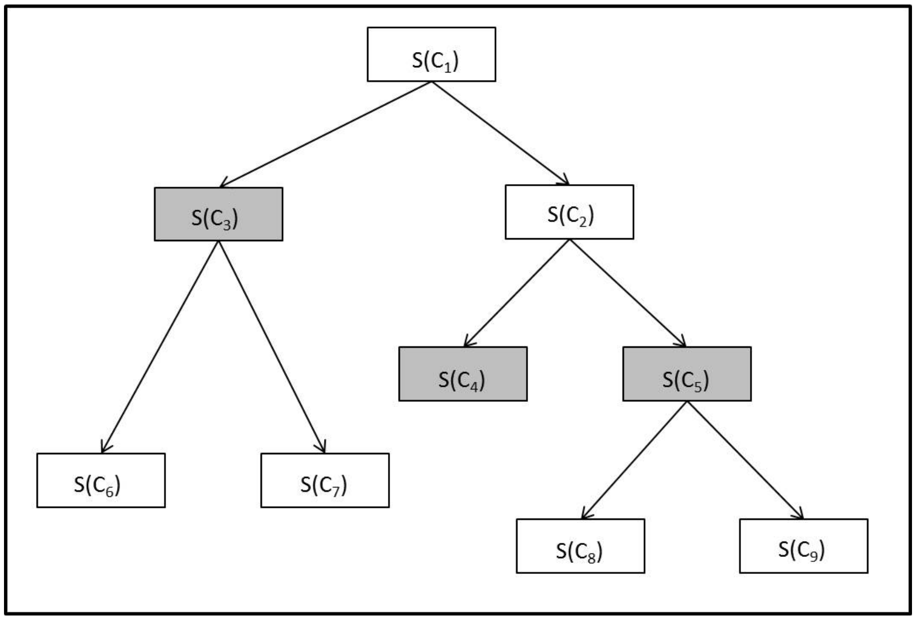

Figure 4.10: Cluster Tree with their respective stabilities

It can be seen that the whole dataset (assuming it to be one cluster, C_1) is initially divided into C_2 and C_3, and these clusters are subsequently divided into other clusters, ultimately forming a set of *leaf clusters* $= \{C_4, C_6, C_7, C_8, C_9\}$. Clusters C_3, C_4 and C_5 are the clusters with maximum area covered in the graph in figure 4.9, which directly correspond to their stabilities. The optimization algorithm starts by first identifying the last formed leaf clusters, adding their stabilities and comparing this sum with their parent. In this case, the clusters, C_8 and C_9 are taken and compared with their parent C_5. Since $S(C_5) > S(C_8)+S(C_9)$, clusters $\{C_8, C_9\}$ are discarded. If the parent's stability is smaller than the sum of stabilities of the children, $S(C_3) < S(C_4)+S(C_5)$, the cluster C_3 is discarded. The whole process is repeated until there are a set of clusters found with the overall maximum stability which is given by $S(C_3) + S(C_4) + S(C_5)$ for the clusters $\{C_3, C_4, C_5\}$ shown in figure 4.9 and 4.10.

Chapter 5

Parallel HDBSCAN*

The HDBSCAN* algorithm has several advantages over traditional partitional clustering algorithm. It combines the aspects of density based clustering and hierarchical clustering, giving a complete density-based hierarchy. It can be visualized through dendrograms or other visualization techniques and does not require an input stating the number of clusters. It is also flexible in that a user can choose to extract clusters as a flat partition or analyze clusters using a cluster tree.

HDBSCAN* algorithm requires the whole dataset to be available in order to calculate the mutual reachability distances between all pairs of data objects to compute the Mutual Reachability Graph, $G_{m_{reach}}$ using which the HDBSCAN* hierarchy is computed. This requirement makes it difficult to parallelize the HDBSCAN* algorithm. With an exponential growth in the amount of data that is generated for analysis through many data centers, it is desirable to have a scalable version of the HDBSCAN* algorithm that will take advantage of parallel systems.

In this chapter, we introduce an algorithm called Parallel HDBSCAN* or PHDBSCAN with two different approaches to compute the HDBSCAN* hierarchy by data parallelism. The first approach called the *"Random Blocks Approach"* is a parallel version of computing the HDBSCAN* hierarchy on parallel and distributed nodes. The HDBSCAN* hierarchy computed using *Random Blocks Approach* is an exact hierarchy that would have been found, had the hierarchy been computed on a single machine. The second approach called *"Recursive Sampling Approach"* is a faster method of computing an approximate version of the HDBSCAN* hierarchy.

5.1 Random Blocks Approach

The HDBSCAN* algorithm can be seen as a generalized version of the Single Linkage (SLINK) Clustering algorithm [13] addressing the main drawback of SLINK called the *"chaining effect"*. A HDBSCAN* hierarchy with $m_{pts} = 2$ will provide the same hierarchy produced by Single Linkage. The Random Blocks Approach is in the same spirit as the algorithm that has been used to parallelize a Minimum Spanning Tree (MST) in [25] by distributing the data to different *"processing units"*. A *"processing unit"* is an independent entity with its own memory and computational resources. It ranges from cores within the same machine to independent and distributed systems on different servers.

The idea of Random Blocks Approach is to find an exact MST of the complete dataset by dividing the complete dataset into blocks of smaller datasets called *"data blocks"*. MSTs (with weights corresponding to mutual reachability distances) are computed on *data blocks* independently and in parallel at different *"processing units"*. These independently computed MSTs are combined to get an MST of the complete dataset. The overall MST is used to compute the exact HDBSCAN* hierarchy of the complete dataset. The algorithm for the Random Blocks Approach is described in Algorithm 4.

The steps explaining the *Random Blocks Approach* is described in Algorithm 4. The flow of the *Random Blocks Approach* is shown in figure 5.1. The diagram assumes that the $m_{pts} = 3$ (for convenience of explaining the diagram) and it is seen that each data block consists of 6 partitions among the k available partitions.

Algorithm 4: Random Blocks Approach

Input: Dataset X, Parameter m_{pts}
Output: HDBSCAN* hierarchy

1. Divide the dataset X into k non-overlapping partitions of roughly equal size, such that $k \geq 2 \times m_{pts}$.
2. Generate $\binom{k}{2\times m_{pts}}$ data blocks, P_l, $1 \leq l \leq \binom{k}{2\times m_{pts}}$, using the k partitions with each data block consisting of $2 \times m_{pts}$ unique partitions.
3. For each data block, compute the local MST (based on mutual reachability graph of the data block) at different processing units.
4. Combine the local MSTs by calling the Algorithm 5 *"Combine MSTs"* to get the overall combined MST.
5. Extend the combined MST to obtain MST_{ext}, by adding a *"self-edge"* for each data object with weight equal to the object's core distance.
6. Extract the HDBSCAN* hierarchy as a dendrogram from MST_{ext}.

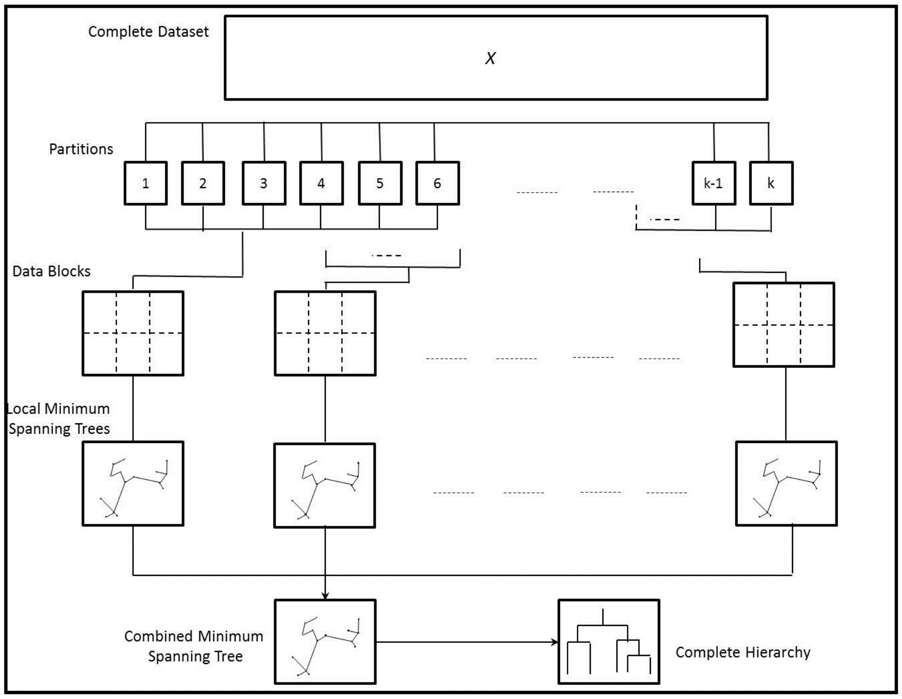

Figure 5.1: Flow of Random Blocks Approach

The steps are explained in detail as follows

5.1.1 Dividing the Dataset

Consider a dataset $X = \{x_i\}$, $1 \leq i \leq n$ which consists of n d-dimensional data objects. The whole dataset is divided into k partitions. The value of k can be chosen based on the following criteria:

1. Select k such that $k \geq 2 \times m_{pts}$ (and)

2. Number of available *processing units.*

The k partitions are given by X'_i, $1 \leq i \leq k$ such that $X = X'_1 \cup, ..., \cup X'_k$

5.1.2 Generating Data Blocks for independent processing

Given a set of k partitions and the parameter m_{pts}, *"data blocks"* are generated by combining partitions. Each *data block* consists of $2 \times m_{pts}$ partitions among the k available partitions. The reason for choosing $2 \times m_{pts}$ partitions per data block is to find the *"true reachability distance"* between a pair of data objects in at least one of the data blocks (explained in detail using Lemma 1 and its proof). Each partition within a data block is unique and no two data blocks consist of the same combination of partitions. By rule of combinations, for k partitions and $2 \times m_{pts}$ partitions per data block, a total of $\binom{k}{2 \times m_{pts}}$ unique data blocks are generated. Each data block is represented by Y_l, where $1 \leq l \leq \binom{k}{2 \times m_{pts}}$.

Before explaining the reason to form data blocks by combining $2 \times m_{pts}$ partitions, we define the following terms

Definition 1: True k-NN neighborhood: Given a data object x_p, a subset Y of a complete dataset X, such that $x_p \in Y$ and $Y \subseteq X$, then Y contains true k-NN neighborhood of x_p, only if

$$m_{pts}\text{-}NN(x_p, Y) = m_{pts} - NN(x_p, X)$$

where m_{pts}-$NN(x_p, X)$ and m_{pts}-$NN(x_p, Y)$ are the set of nearest neighbors of x_p in datasets X and Y respectively. This means that Y should contain all the m_{pts} neighbors of x_p in X.

Definition 2. True Core Distance: True Core Distance for a data object x_p, $d_{true\text{-}core}(x_p)$, w.r.t m_{pts} is defined as the distance to its m_{pts}-Nearest Neighbor in their *true m_{pts}-NN neighborhood.*

Definition 3. True Mutual Reachability Distance: The *"true mutual reachability distance"* between two objects x_p and x_q is given by

$$d_{true\text{-}mreach}(x_p, x_q) = max\big(d_{true\text{-}core}(x_p), d_{true\text{-}core}(x_q), d(x_p, x_q)\big)$$

To find the *true mutual reachability distance* of a data object, it is required to have at least $2 \times m_{pts}$ partitions in each data block. The guarantee to find the *"true mutual reachability distance"* is explained in proof of Lemma 1 as follows:

Lemma 1: Given $\binom{k}{2 \times m_{pts}}$ data blocks generated using k non-overlapping partitions of a dataset X, there is a guarantee that at least one data block contains the *"true mutual reachability distance"* w.r.t. m_{pts} for a pair of data objects.

Proof: Consider two data objects x_p and x_q belonging to one of the data blocks Y_l but different partitions, X'_p and X'_q respectively. In order to find the *"true mutual reachability distance"* between x_p and x_q, we need the *true core distances* for both x_p and x_q. Given the worst case scenario where the m_{pts} neighbors of both x_p and x_q belong to different partitions, such that no two objects in the set $m_{pts}\text{-}NN(x_p) \cup m_{pts}\text{-}NN(x_q)$ belong to the same partition, then there is at least one data block composed of $2 \times m_{pts}$ partitions with each partition containing one object of the set $m_{pts}\text{-}NN(x_p) \cup m_{pts}\text{-}NN(x_q)$ (since $\binom{k}{2 \times m_{pts}}$ contains all possible combinations). This property guarantees that there is at least one data block for each $x_p, x_q \in X$ such that $d_{true\text{-}mreach}(x_p, x_q)$ can be computed. ■

5.1.3 Sending Data Blocks for parallel processing

There is a *"master processing unit"* that orchestrates the processing by all other processing units and serves the data blocks to them. Each processing unit receives a data block for further processing and the procedure of sending the data blocks depends on the framework used for implementation.

5.1.4 Computation of an MST at a Processing Unit

Each data block, Y_l, is sent to a *processing unit.* At a processing unit, compute the core distances w.r.t m_{pts} for all data objects in Y_l and form the Mutual Reachability Graph, $G_{m_{reach}}$. A Minimum Spanning Tree, $MST_{local}(Y_l)$ is then constructed on $G_{m_{reach}}$. The $MST_{local}(Y_l)$ is returned to the *master processing unit.*

Algorithm 5: Combine MSTs

```
Input: Set of $\binom{k}{2\times m_{pts}}$ local MSTs
Output: Single combined MST
  1. Add all the edges from all the local MSTs to form a set of edges,
     $E_{local}$.
  2. Sort all the edges in $E_{local}$ ascendingly based on their edge weights.
  3. Initialize $G_{MST}$ for storing the combined MST. Currently, $G_{MST}$ does
     not contain any edges.
  4. While $E_{local} \neq \emptyset$
     (a) Remove the edge $e$ with the least weight in $E_{local}$ and add it to
         $G_{MST}$.
     (b) If an edge $e'$ in $G_{MST}$ has the same vertices as that of $e$, then
         update $w(e') = min\{w(e'), w(e)\}$ and discard $e$.
     (c) If addition of $e$ to $G_{MST}$ causes a loop in $G_{MST}$, remove the edge
         from $G_{MST}$.
  5. Return $G_{MST}$, the combined MST
```

5.1.5 Combining Local MSTs from different processing units

Once all the local MSTs constructed at different processing units have been received by the *master processing unit*, the next step is to combine them to get the MST of the complete dataset. The algorithm to combine the MSTs is explained in Algorithm 5: "Combine MSTs".

Consider all the edges from all the local MSTs and add them to a set of edges, E_{local}. Sort all the edges in E_{local} in ascending order of their weights. Initialize $MST_{combined}$ that would maintain the overall combined MST of the complete dataset. Remove the edge e with the least weight from E_{local} and add it to $MST_{combined}$. If there is another edge e' in $MST_{combined}$ with the same vertices as that of edge e, update the edge weight of e' with the minimum edge weight of e and e' and discard e. If edge e creates a loop in $MST_{combined}$, remove e from $MST_{combined}$. Duplicate edges are possible since many data blocks contain overlapping data objects, translating to common set of nodes in many local Mutual Reachability Graphs. Different edge weights are possible for edges with the same end vertices because all data blocks do not contain the *true mutual reachability distance* as edge weight. Minimum value is chosen for the edge weight to be updated, since the *true mutual reachability distance* is the least possible weight for an edge, as per Lemma 2. According to Lemma 1, there will be at least one edge with weight corresponding to the *true mutual reachability distance*. The process is continued until all the edges in E_{local} are removed. The $MST_{combined}$ thus constructed is the MST of the complete dataset.

Lemma 2: Given two data objects, x_p and x_q, with a list of m measures of their mutual reachability distances $D = \{d_1, ..., d_m\}$ obtained for the pair of object (x_p, x_q) from m different processing units, the true mutual reachability distance between x_p and x_q is the minimum distance in the set D.

$$d_{true\text{-}mreach}(x_p, x_q) = min\{d_1, ..., d_m\}$$

Proof: According to Definition 2, it is known that the *true core distance* of any data object x_p is the distance to its m_{pts}-Nearest Neighbor in their *true m_{pts}-NN neighborhood.* Consider a set of data objects $Y \subset X$, with the m_{pts}-neighborhood of x_p in sets X and Y are given by m_{pts}-$NN(x_p, X)$ and m_{pts}-$NN(x_p, Y)$ respectively. If m_{pts}-$NN(x_p, X) \neq m_{pts}$-$NN(x_p, Y)$, then m_{pts}-NN neighborhood of x_p in Y is not a true m_{pts}-neighborhood. By definition of nearest neighbor, there exists at least one object, x_r in m_{pts}-$NN(x_p, Y)$ that is at a distance, $d \geq d_{true\text{-}core}(x_p)$. Therefore, the core distance of x_p w.r.t m_{pts} in dataset Y, $d_{core}(x_p, Y) \geq d_{true\text{-}core}(x_p)$. For any given set of core distances for l different subsets of X, $d_{true\text{-}core}(x_p)$ is given by

$$d_{true\text{-}core}(x_p) = min\{d_{core}(x_p, Y_1), ..., d_{core}(x_p, Y_l)\} \tag{5.1}$$

Similarly for an object x_q, the *true core distance* is given by

$$d_{true\text{-}core}(x_q) = min\{d_{core}(x_q, Y_1), ..., d_{core}(x_q, Y_l)\} \tag{5.2}$$

and $d(x_p, x_q)$ is always a constant irrespective of the partitions to which x_p and x_q belong. By definition of *true mutual reachability distance*, a true mutual reachability distance is computed based on the *true core distances* of x_p and x_q. By combining equations 5.1 and 5.2 and the property of $d(x_p, x_q)$, it can be concluded that for a given set of mutual reachability distances between objects x_p and x_q, true mutual reachability distance is given by

$$d_{true\text{-}mreach}(x_p, x_q) = min\{d_1, ..., d_m\}$$

■

It is required to prove that the constructed tree is a global Minimum Spanning Tree. In order to prove this, the following two lemmas are provided

Lemma 3: The output of the combination of all local MSTs after pruning is a spanning tree.

Proof: During the merging procedure, while adding edges to the final global MST, $MST_{combined}$, no edges creating cycles are included in $MST_{combined}$. All duplicate edges are also pruned in the process. The only other requirement is to show whether all the vertices have been added to the global MST. The output is a combination of all MSTs generated from the data blocks Y_l which is composed of subsets given by X'_i, $1 \leq i \leq k$ so that $X = X'_1 \cup, ..., \cup X'_k$. Each partition is therefore available in at least one of the data blocks and local MSTs are created based on the complete mutual reachability graph corresponding to all the objects in the data block. Therefore, vertices corresponding to all nodes are available in $MST_{combined}$. ■

Lemma 4: If $M(G)$ is the global spanning tree of the complete dataset X by combining all the local MSTs, then $M(G)$ is the global Minimum Spanning Tree.

Proof: Assume that $MST(G)$ is a global Minimum Spanning Tree of X and let w($MST(G)$) denote the total weight of the MST. Since w($MST(G)$) is the minimum weight of a spanning tree for X, any other spanning tree T must satisfy the condition $w(MST(G)) \leq w(T)$. We will show by contradiction that $w(M(G)) \leq w(MST(G))$.

Let us assume $w(M(G)) > w(MST(G))$ and let $E(MST(G))$, $E(M(G))$ be the set of mutually exclusive edges of the graph $MST(G)$ and $M(G)$ respectively. This implies that $E(MST(G)) \cap E(M(G)) = \emptyset$. As spanning trees of graph G, $MST(G)$ and $M(G)$ must have the same number of edges (since the number of vertices are equal), and the number of edges in sets $E(MST(G))$ and $E(M'(G))$ are equal. Thus the weights of the MSTs, $MST(G)$ and $M(G)$ are different. There is at least one pair of edges with different weights that are not common in $MST(G)$ and $M(G)$. Therefore, $E(MST(G))$ and $E(M'(G))$ must be non-empty.

Let $e = (x_p, x_q)$ be an arbitrary edge of $E(MST(G))$. Adding e to the spanning tree $M(G)$ will induce a single cycle in $M(G)$ and there would be a unique path $(x_p, x_q) \neq e$ in $M(G)$. Let E_h be the set of edges in this path, not contained in $MST(G)$. Since $MST(G)$ contains e but not the complete cycle, E_h is not empty. If there was always some $e' \in E_h$ such that $w(e') \leq w(e)$, $M'(G) = M(G) - \{e'\} \bigcup \{e\}$ would be a spanning tree with $w(M'(G)) \geq w(M(G))$ that has one more edge in common with $MST(G)$, and since $MST(G)$ and $M(G)$ are finite, $M(G)$ can be converted into $MST(G)$ without reducing its weight, contradicting our assumption that $w(M(G)) > w(MST(G))$. Thus, if $w(M(G)) > w(MST(G))$, there must be some such set E_h and edge e with $w(e') > w(e)$ for all $e' \in E_h$.

Next, we will prove by contradiction that the set $E_h = \emptyset$, by showing that the (x_p, x_q)-path in $M(G)$ must not contain any edge of E_h. Let $e = (x_p, x_q)$. As every pair of data objects are distributed to some data block along with their *true m_{pts}-NN neighborhood*, there will be some local MST generated by processing unit containing e which includes the path (x_p, x_q) with weight no more than $w(e)$. When the MSTs of different data blocks are merged, a path $e' = (x_p, x_q)$ from a different MST may be removed from $MST_{combined}$ result if there is some (x_p, x_q) in the current solution that has weight no more than $w(e)$. The path e', is not added to E_h. As this observation holds true for every step of the merging procedure, E_h will never contain any path (x_p, x_q) such that $w(e') \leq w(e)$ and the final merged output, $M(G)$ must contain a (x_P, x_q)-path composed of edges with weight no more than $w(e)$. Also, since $w(e') > w(e) \forall e' \in E_h$, this (x_p, x_q)-path cannot contain any edge of E_h, contradicting the initial definition of E_h as a set of edges along the (x_p, x_q)-path in $M'(G)$. ■

5.2 Recursive Sampling

The second approach to parallelize the HDBSCAN* algorithm is called *"Recursive Sampling Approach"*. Unlike the *Random Blocks Approach*, the *Recursive Sampling Approach* eliminates the processing of overlapping datasets at multiple processing units. This is achieved by *"intelligent"* data division, which divides the data to be processed at different processing units based on the structure of the data.

For a data object not to be duplicated across multiple nodes, it is required that the overall coarse structure of the data is captured and the data blocks are created according to the data divided on this coarse structure. The data blocks are sent to different processing units for further refinement. Depending on the capacity of the processing units, the data blocks can be recursively divided into smaller data blocks until the data blocks are of sizes that can be processed by a processing unit.

The algorithm for *"Recursive Sampling Approach"* is shown in Algorithm 6.

```
Algorithm 6: Recursive Sampling
  Input: Dataset $X$, Parameters $m_{pts}$ and $m_{clSize}$, Processing Capacity $\tau$,
         Division Method $div$
  Output: HDBSCAN* hierarchy
    1. Call Algorithm 7: Recurse($X, m_{pts}, m_{clSize}, div$) to get a set of
       edges $E_{MST}$ and $E_{inter}$.
    2. Call Algorithm 8: Combine ($E_{MST}, E_{inter}$) to get an MST,
       $MST_{combined}$.
    3. Construct HDBSCAN* hierarchy as a dendrogram from
       $MST_{combined}$.
```

The algorithm of *Recursive Sampling Approach* can be broadly divided into two major steps, (i) *Recurse* step and (ii) *Combine* step. The *Recurse* step intelligently divides the datasets into smaller datasets based on the structure of data. The *Recurse* step returns two sets of edges, E_{MST} and E_{inter}. Set E_{MST} contains edges of all the local MSTs and the second set E_{inter} contains the *inter-connecting edges*. Inter-connecting edges are the shortest edges between different components identified. The *Combine* step combines all the edges to form the MST of the complete dataset using which the HDBSCAN* hierarchy can be extracted.

Algorithm 7: Recurse

```
Input: Dataset $Y$, Processing Capacity $\tau$, Parameter $m_{pts}$, Division
       Method $div$, Sampling Tolerance $\varsigma_{max}$
(optional) Output: Set of Local MSTs, Set of Inter-connecting Edges
  1. Call the Algorithm 10: Divide($Y$, $m_{pts}$, $div$, $\varsigma_{max}$) to get a set of $k$
     components $\{Y_1, ..., Y_k\}$ and $E_{inter}$. If "fail" was returned from Divide,
     return ;

  2. foreach $Y_C \in \{Y_1, ..., Y_k\}$ do
       if $\mid Y_C \mid \leq \tau$ then
         Call Algorithm 11: Compute MST($Y_C$) to get $MST_{local}$ and
         $E_{inter}$.
         return $MST_{local}$ and $E_{inter}$.
       else
         Call Algorithm 7: Recurse($Y_C, \tau, m_{pts}$).
         If "fail" was returned, goto step 1
       end
     end
```

A step by step approach of an example dataset is illustrated using figures 5.2 through 5.11. Figure 5.2 shows the flow of Recursive Sampling Approach. Figure 5.3 shows an example dataset which consists of three large clusters and each cluster is recursively divided into sub-clusters.

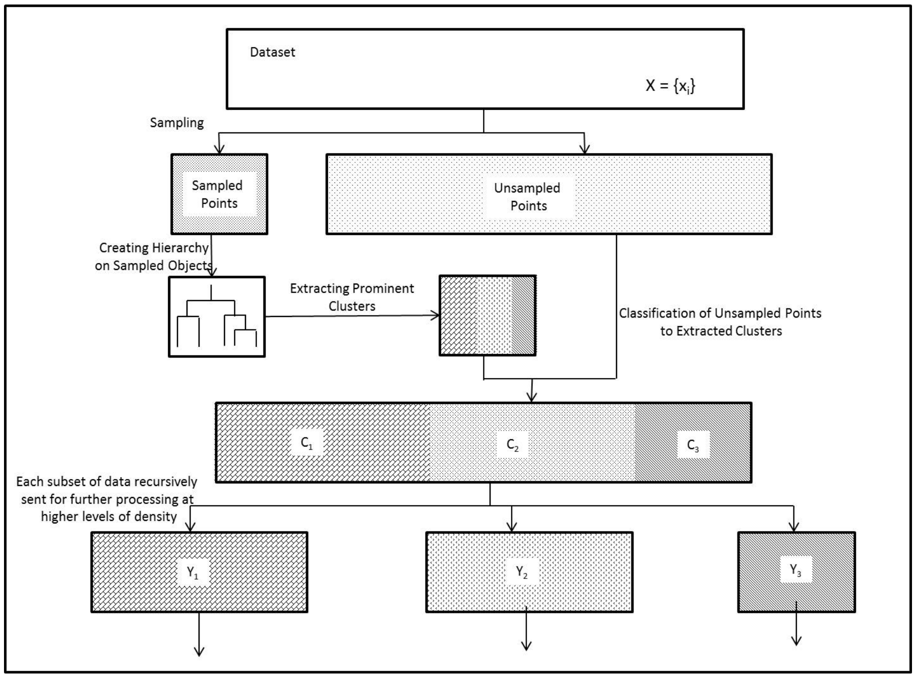
Dataset
X = {x_i}
Sampling
Sampled Points
Unsampled Points
Creating Hierarchy on Sampled Objects
Extracting Prominent Clusters
Classification of Unsampled Points to Extracted Clusters
C1
C2
C3
Each subset of data recursively sent for further processing at higher levels of density
Y1
Y2
Y3

Figure 5.2: Flow of Recursive Sampling approach

Given a dataset X with n data objects, a random sample S of m objects is drawn from X. The value of m should be high enough to reflect the coarse structure of the dataset. The sampled objects are given by

$$Sampling_{random}(X) = S = \{s_j\}; 1 \leq j \leq m$$

The unsampled data objects belong to a set US such that $US = X - S$. Sampled objects of the dataset in figure 5.3 are shown in figure 5.4.

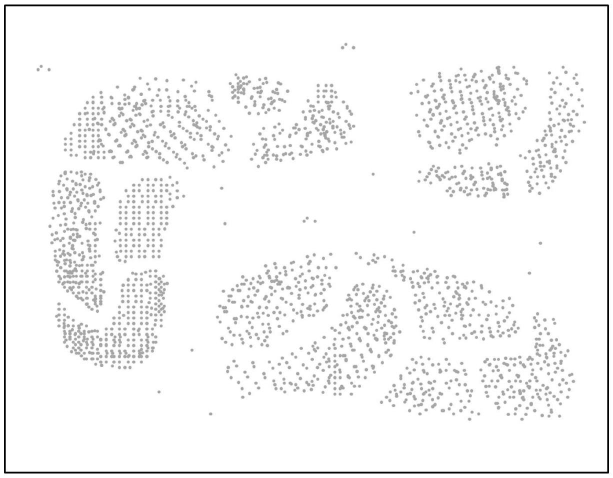

Figure 5.3: An example Dataset

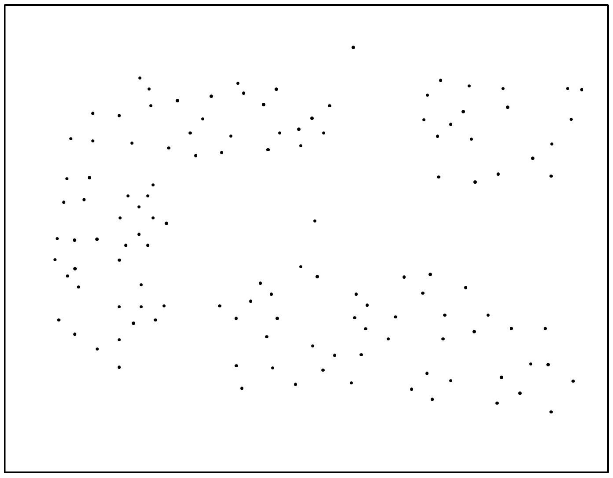

Figure 5.4: A set of sampled objects of dataset shown in figure 5.3

Algorithm 8: Combine

```
Input: Set of all local MSTs $M$, Set of Inter-connecting edges $E$
Output: Overall HDBSCAN* hierarchy
  1. Initialize set $E_{combined}$.
  2. foreach $M' \in M$ do
         Add all edges in $M'$ to $E_{combined}$.
     end
  3. foreach $E' \in E$
         Add all edges in $E'$ to $E_{combined}$.
     end
  4. Set $E_{combined}$ consists of all edges that form the MST, $MST_{combined}$, of
     the complete dataset.
  5. Extract the HDBSCAN* hierarchy using $MST_{combined}$.
```

Cluster Extraction: Calculate the core distances for all objects in S w.r.t m_{pts} and compute the Mutual Reachability Graph, $G_{m_{reach}}$. Construct a minimum spanning tree MST_{local} on $G_{m_{reach}}$ and extract HDBSCAN* hierarchy from MST_{local}. The most prominent clusters are extracted from HDBSCAN* hierarchy in one of the following two ways:

1. **Binary Extraction:** This is a simple version of extracting the clusters where the first formed clusters are immediately extracted. Binary extraction is fast since it does not have to traverse through the complete hierarchy for the first formed clusters to be identified.

   Initially, the edges in $MST(G_{m_{reach}}(S))$ are sorted in descending order of their edge weights. Edge with the largest edge weight is removed (if there are multiple edges with the same edge weights, they are removed simultaneously) to form two or more disconnected components. After the removal of any given edge(s) of weight ϵ, if there are two or more disconnected components which form prominent clusters (given by $\mid component_i \mid \geq m_{clSize}$), then the data objects belonging to these components are assumed to be prominent clusters.

This means, for any given dataset, if there is a clear demarcation between two clusters, this approach will identify it very quickly and distribute the dataset to different processing units for further refinement. The Binary extraction method applied on the dataset in figure 5.3 is shown in figure 5.5, which shows that the sampled data objects are divided into two clusters, C_A and C_B. The algorithm for Binary Extraction is explained in algorithm 9.

```
Algorithm 9: Binary Extraction
  Input: Minimum Spanning Tree $M$, Parameter $m_{clSize}$
  Output: Set of Clusters $C = \{C_1, ..., C_k\}$
    (a) Initialize set $C = \emptyset$ and a tree $T$
    (b) Sort the MST $M$ in descending order of weights of its edges
    (c) while edges exist in $M$ do
            i. Remove the edge(s) with the maximum weight from $M$.
           ii. Find the set of connected components $CC$.
          iii. If (there are at least two components in $CC$ such that their
                 size $\geq m_{clSize}$) then
                 Add components with size $\geq m_{clSize}$ to set $C$;
                 return;
               end
        end
```

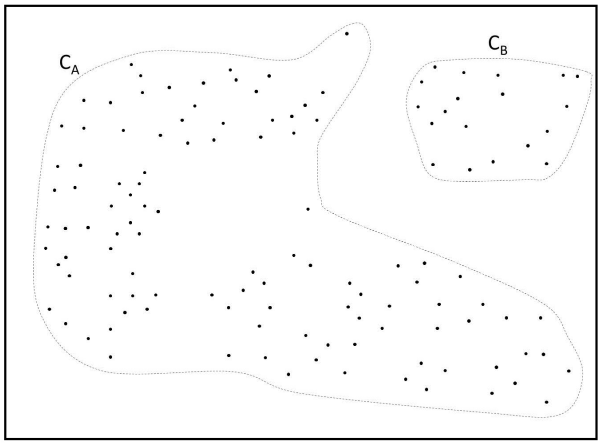

Figure 5.5: Illustration of Binary Extraction of Clusters shown in figure 5.4

2. **N-ary Extraction:** This approach of extracting the prominent clusters traverses through the complete hierarchy, by removing the edges with the maximum edge weight, until all edges have been removed and then forms a cluster tree. Once the cluster tree has been formed, the most prominent clusters are extracted in the form of non-overalapping cluster sets on the basis of excess of mass explained in section 4.6. The final result is a set of clusters $\{C_1, ..., C_k\}$ with each cluster containing objects from S. The N-ary extraction method applied on the dataset in figure 5.3 is shown in figure 5.6, which shows that the sampled data objects are divided into three clusters, C_A, C_B and C_C.

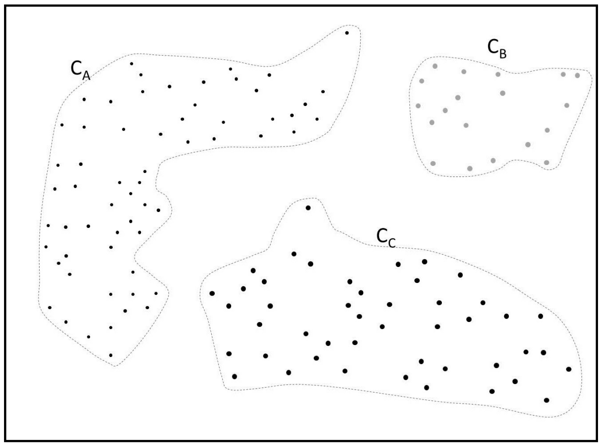

Figure 5.6: Illustration of N-ary Extraction of Clusters

Inclusion of Noise objects to current clusters: It is evident from the HDBSCAN* cluster extraction, that not all data objects in dataset S are assigned to a cluster. Objects in S that are not included in one of the extracted clusters are called *"noise"* objects and are added to the set O.

$$S = (C_1 \cup ... \cup C_k) + O$$

$$\mid S \mid = \Big(\sum_{i=1}^{k} \mid C_i \mid \Big) + \mid O \mid$$

The HDBSCAN* hierarchy is built based on the sampled objects and the objects in O are *"noise"* objects only with respect to the sampled set of objects and not with respect to the complete dataset. In order to reduce data loss, these noise objects are added back to one of the clusters.

The data objects in set O are added back to hierarchy in the following way:

1. For each object $x_o \in O$, find the nearest neighbor of x_o in $(C_1 \cup ... \cup C_k)$. The nearest neighbor is denoted by 2-$NN(x_o)$.

2. Find the cluster membership of the 2-$NN(x_o)$, among the set of clusters identified, $C_1, ..., C_k$.

$$C_o = member(2\text{-}NN(x_o))$$

3. Add the data object x_o to the cluster C_o.

$$C_o = C_o \cup \{x_o\}$$

Classification of unsampled objects: The set of unsampled data objects are assigned to one of the clusters which contains the nearest neighbor of the data object (similar to how the noise objects were classified to the clusters).

For each object x_u in the set US, find the cluster membership of the nearest neighbor of x_u, $C_u = member(2\text{-}NN(x_u))$. Add x_u to the cluster C_u, given by $C_u = C_u \cup \{x_u\}$. The clusters that are formed are mutually exclusive and there is no common data object between any two given clusters.

$$C_i \cap C_j = \emptyset,\ 1 \leq i, j \leq k;\ i \neq j$$

The first level of recursion has been completed. The result for the dataset shown in figure 5.3 is shown in figure 5.7 for both Binary and N-ary extraction methods. Binary methods consists of only first two found clusters, $\{C_A, C_B\}$. whereas the N-ary method consists of 3 most prominent clusters, $\{C_A, C_B, C_C\}$. Each component is sent to a different processing unit for further refinement. Figures 5.8 through 5.11 shows how the data of Cluster C_A is divided into components (sub-clusters) in the next recursive level. Figure 5.8 shows the sampled objects of Cluster C_A and the clusters that were formed for C_A at a processing unit. Each component here is the data block which is sent to 3 other processing units for refinement. The recursion continues until each processing unit receives data blocks, which can be processed without sampling, i.e., the data block is small enough for the HDBSCAN* hierarchy to be computed on the data block by a single processing unit with an acceptable execution time (depending on the application and the hardware used).

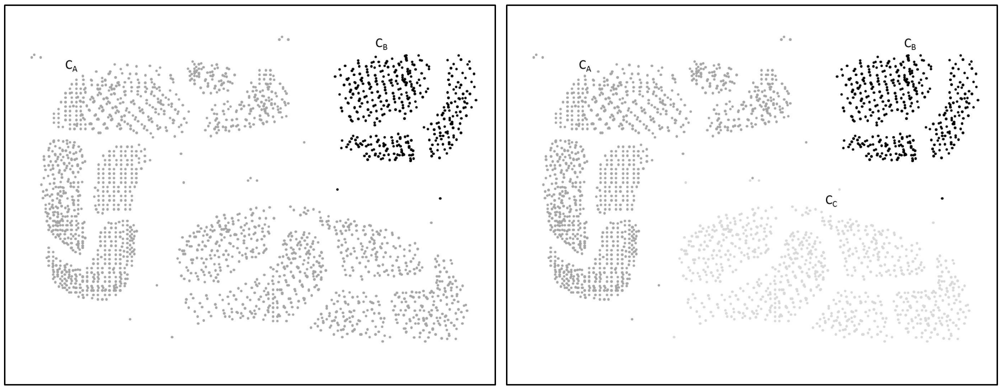

Figure 5.7: Illustration of clusters returned by Binary and N-ary Cluster Extraction methods

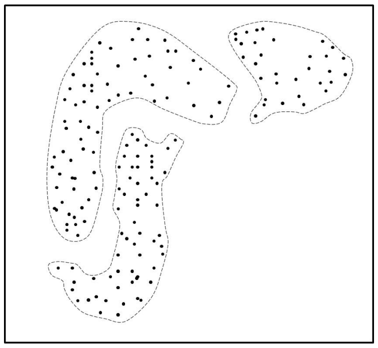

Figure 5.8: N-ary Cluster Extraction on Sampled Objects of cluster C_A from figure 5.6

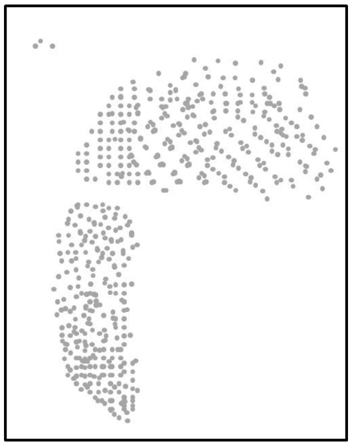

Figure 5.9: All objects of cluster C_Y

Algorithm 10: Divide

```
Input: Dataset $Y$, Parameter $m_{pts}$, Division Method $div$, Sampling
       Tolerance $\varsigma_{max}$
Output: Set of Divided Components: $\{Y_1, ..., Y_k\}$, Set of
        Inter-connecting Edges $E_{inter}$
  1. Draw a sample $S$ of size $m$ from $Y$. The unsampled objects belong
     to the set $US = Y - S$.
  2. With dataset $S$ of $m$ $d$-dimensional data objects, construct a
     HDBSCAN* hierarchy:
     (a) Construct HDBSCAN* hierarchy w.r.t. $m_{pts}$ based on the
         Mutual Reachability Graph, $G_{m_{reach}}(S)$. $G_{m_{reach}}(S)$ is a
         complete graph constructed on dataset $S$.
     (b) Extract prominent clusters from the constructed HDBSCAN*
         hierarchy, $C_S = \{C_{s_i}\}$, $1 \leq i \leq k$ along with a set of "noise"
         objects, $O$. Clusters are extracted using different approaches
         based on the Division Method $div$.
         if $div = Binary$
            Call Algorithm 9 BinaryDivision (HDBSCAN*
            hierarchy)
         else
            Extract a flat-partitioning of most prominent clusters from
            HDBSCAN* hierarchy using the concept of Excess of Mass
            as shown in Algorithm 3.
  3. Classify the noise objects in $O$ into one of the existing clusters by
     finding the cluster label of the nearest neighbor in the set $S - O$ of
     each object in $O$.
  4. For each object $x_s$ in the set $US$, find the nearest neighbor of $x_s$ in $S$
     and classify $x_s$ to the same cluster as its nearest neighbor, resulting in
     $Y = \{Y_{C_1} \cup ... \cup Y_{C_k}\}$, where $Y_{C_i}$ is the set of data objects with cluster
     label $C_i$.
  5. Add the edges in $MST_{sample}$ that connect objects in different clusters
     to $E_{inter}$.
  6. Check for sampling tolerance $\varsigma$. If $\varsigma > \varsigma_{max}$, return "fail" message to
     the calling function.
```

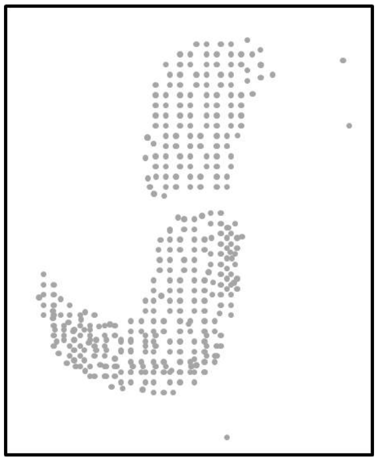
Figure 5.10: All objects of cluster C_X

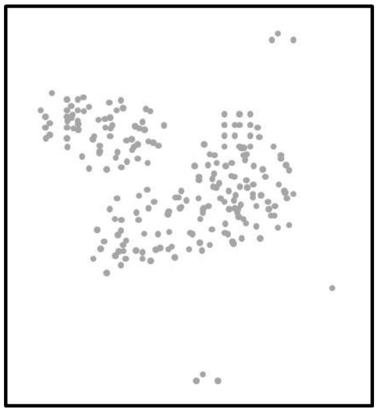
Figure 5.11: All objects of cluster C_Z

Algorithm 11: Compute local MST and inter-cluster edges

```
Input: Dataset $Y$
Output: Minimum Spanning Tree $M_{local}$, Set of inter-connecting edges
        $E_{inter}$
    1. Calculate core distances of all data objects in $Y$ w.r.t $m_{pts}$.
    2. Compute the Mutual Reachability Graph, $G_{m_{reach}}(Y)$.
    3. Compute an MST ($MST_{local}$) on $G_{m_{reach}}(S)$.
    4. Construct HDBSCAN* hierarchy based on the MST formed.
    5. Extract a flat-partitioning of most prominent clusters from
       HDBSCAN* hierarchy as shown in Algorithm 3.
```

Finding Inter-Cluster Edges: Before the data objects are sent to different processing units for refinement based on the clusters to which they belong, it is required to find the edges between different clusters. These edges are called as *"inter-cluster edges"* or *"inter-connecting edges"*. Set of *inter-cluster edges* play an important role in maintaining the link between different clusters and to complete the overall combined MST which is constructed by combining the edges of many local MSTs.

Randomly selecting a vertex, each from a cluster C_i and C_j and considering them as inter-cluster edge is not a good approximation of distance between two clusters. We propose two different strategies for finding the inter-connecting cluster that estimate the distance between the clusters.

1. **Using already existing edges:** This approach estimates the shortest edge between clusters based only on sampled objects. Every node in the MST formed on the sample set S is transitively reachable to every other node. This property is used to find the edges that will connect different clusters. Below are the steps shown to identify the *inter-cluster edges*:

   (a) Re-label all the nodes in MST_{local} with the cluster label of objects corresponding to the nodes in MST_{local}. Each node is a cluster identified using cluster label and MST_{local} becomes a multigraph (graph where each pair of nodes is permitted to have more than one edge).

   (b) Construct a minimum spanning tree MST_{inter} on this multigraph by eliminating all the self-edges (edges with same nodes in both ends) and any possible duplicate edges by keeping only edges with the minimum weight.

   (c) Add edges in MST_{inter} to E_{inter}. The edges in E_{inter} are the *inter-cluster edges.*

   An example of inter-cluster edges (shown as solid lines) for the dataset shown in 5.3 extracted from the Minimum Spanning Tree, with clusters extracted using N-ary method is shown in figure 5.12.

2. **Estimating the shortest edges using all objects:** This method can be used with only shared memory architecture since maintaining data structures across distributed nodes will increase the communication cost considerably. All unsampled objects are added to one of the clusters by finding the cluster membership of their nearest neighbor. To find the nearest neighbor, each unsampled object is compared with all the objects in S. These distances can be used to estimate the shortest distance and the corresponding edge between two clusters. The steps are as follows:

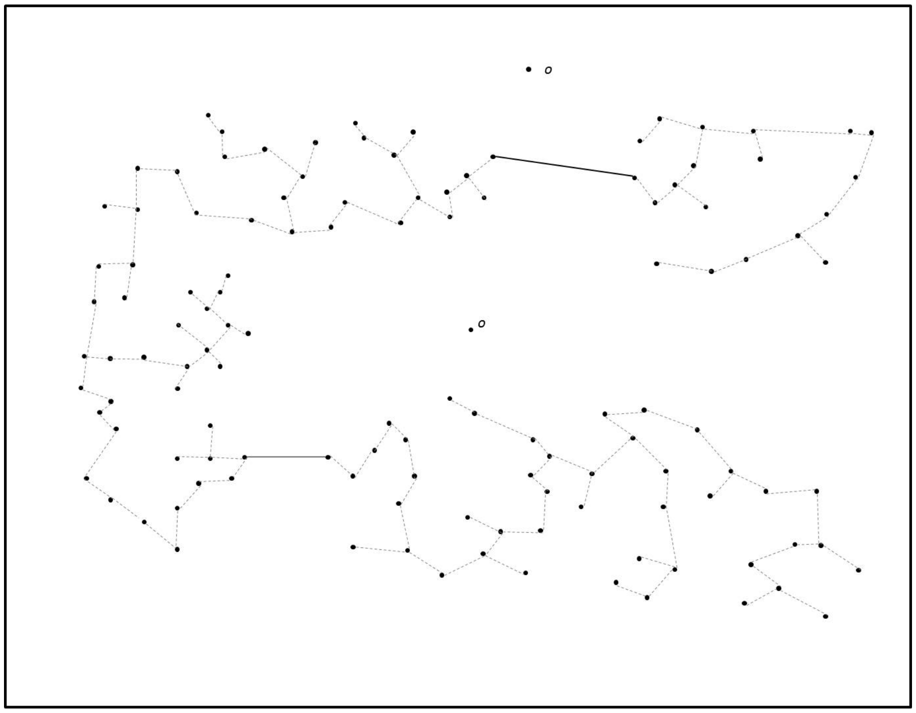

Figure 5.12: Illustration of inter-cluster edges found sampled objects shown in figure 5.4 using the N-ary method of cluster extraction

(a) Initialize a set D to store the shortest edge between every pair of clusters. $D = \{e_{short}(C_i, C_j)\},\ 1 \leq i, j \leq k;\ i \neq j$.

(b) For each object x_u in set US, while finding the nearest neighbor in S, whose cluster memberships are already known, store the shortest edge to each cluster in a set $D' = e_{short}(x_u, C_j),\ 1 \leq j \leq k$.

(c) After assigning x_u to a cluster C_u which consists of 2-$NN(x_u)$, update all the edges $e_{short}(C_i, C_j)$ in set D with corresponding edges in D' such that $i = u$ and $1 \leq j \leq k$ if the weight of edge in D' is smaller than the weight of edge in D.

(d) Construct an MST based on the edges in D to get the inter-cluster edges.

The inter-cluster edges are sent to the calling function when this procedure returns.

Dividing the data and sending it to different processing units: Data objects that belong to the same cluster are considered to form a *"data block"*. The number of data blocks created is equal to the number of most prominent clusters that are formed at any recursive step. Data blocks are given by a set of mutually exclusive datasets $\{Y_1, ..., Y_k\}$. Each data block is sent to a different processing unit for further refinement. In the processing unit, if the overall size of the data block is greater than the capacity of the processing unit, τ, the steps are repeated by calling the **Algorithm 7: Recurse** with Y_i as the dataset. The capacity of the processing unit might vary from system to system depending on the hardware and processing power of the unit. The allocation of a data block to a processing unit depends on the architecture of the multi-core or multi-node system on which the algorithm runs.

Optional Sampling Tolerance check: The sampling tolerance check is a way of finding if the hierarchy based on the sampled objects reflected the overall structure at least at a coarse level. It is an optional parameter and can identify discrepencies in the hierarchy. Consider the subset of data shown in figure 5.13, with sampled objects a and b, each belonging to clusters A and B respectively. All other objects are assigned to the nearest object among a and b. The distance between clusters A and B is given by the distance e_{AB}. This distance reflects the density level λ_{AB} at which clusters A and B are created. The cluster A is passed on to a different processing unit, for futher refinement and similar scenario is repeated with x and y as the sampled objects belonging to clusters C and D with other objects are assigned to one of the two sampled objects. Let e_{CD} denote the weight of the inter-cluster edge between C and D and gives the density at which clusters C and D appear. If $e_{CD} > e_{AB}$, it indicates that the cluster A was born at a higher density level and its children, clusters C and D were born at a lower density level than A. This scenario occurs if the sampled set is skewed and not distributed uniformly.

This issue could be immediately addressed at Processing unit 2 and the hierarchy can be repaired by resampling the dataset available at Processing Unit 1. Addition of unsampled data objects and a different sample set at the next

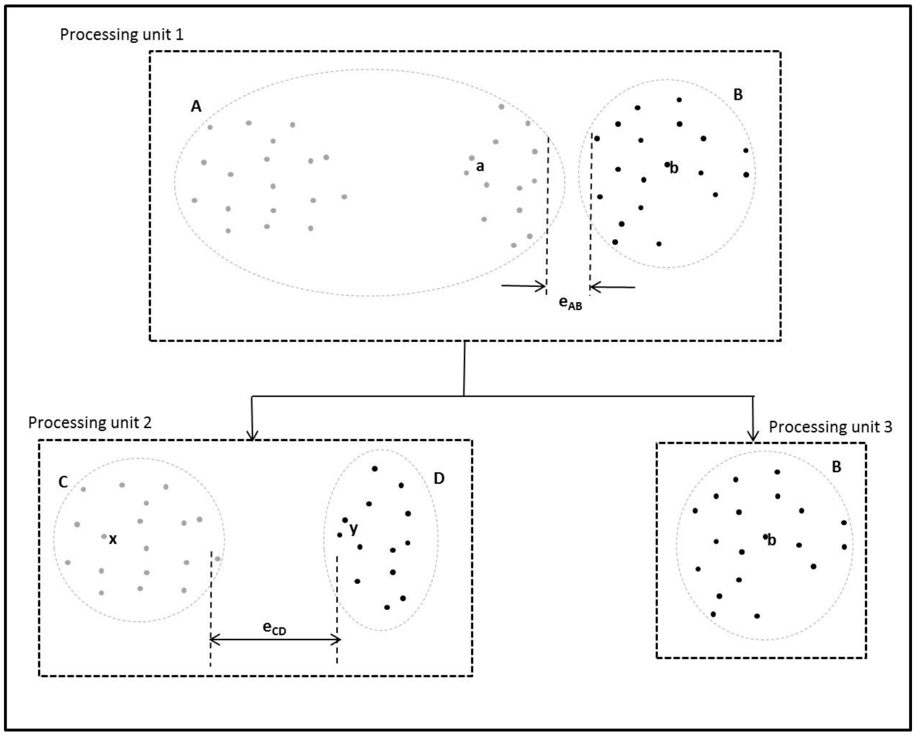

Figure 5.13: Situation where the parent is born at a higher density level than the density of its disappearance

level can induce such errors which can be controlled by a parameter ς. The sampling tolerance parameter gives the notion of ratio of the e_{CD} to e_{AB}. A sampling tolerance value of $\varsigma = 1$ is a strict sampling tolerance since it does not allow any child cluster to be created at a lower density than its parent. If there are multiple children with multiple edges in a set E_{inter} between them, then ς is compared to every edge in E_{inter}.

Chapter 6

Data Summarization using Data Bubbles

6.1 Drawbacks of building hierarchy based only on Samples

There are several drawbacks of constructing a HDBSCAN* hierarchy based on sampled objects. They are explained as follows:

1. **Capturing the structure:** Since the dataset that is used to build the HDBSCAN* hierarchy is based on a set of sampled objects, there is no guarantee that the overall structure of the data is captured. This directly affects the structure of the final hierarchy.

2. **Estimating the inter-cluster edges:** The shortest edge between two clusters is considered to be an inter-cluster edge. While using the PHDB-SCAN on sampled objects, the inter-cluster edges are estimated by only using the set of sampled objects belonging to a cluster. But, better estimates of the weight of inter-cluster edges can be identified if the weights are calculated based on some inexpensive technique involving the complete dataset and not just the set of sampled objects.

3. **Re-inclusion of "Noise" objects:** Including *noise* objects back to the dataset until the final level of recursion may affect the quality of intermediate hierarchies constructed at distributed nodes.

Consider the datasets shown in figure 6.1 and 6.2 where each dataset has a set of data objects with the sampled data objects $\{x, y, z\}$. Let us assume that the sampled data object z has been identified as a noise object while extracting the clusters from the HDBSCAN* hierarchy on the dataset. Figure 6.1 shows that the density around object z is relatively high while in figure 6.2 the neighborhood of the sampled object z is relatively less dense. The density around z is not known during cluster extraction since the extraction is based only on the sampled objects. Re-including noise objects that are not dense will propagate the noise objects to all the levels of recursion, and these noise objects will be available until the final overall hierarchy is computed.

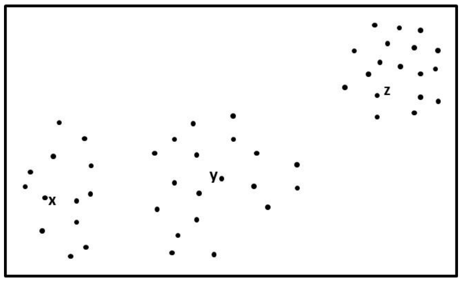

Figure 6.1: Example where one of the sampled object z is dense

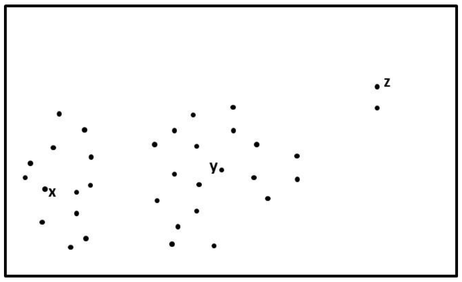

Figure 6.2: Example where one of the sampled object z is not dense

4. **Problems pertaining to the parameter m_{clSize}:** The parameter m_{clSize} is used when extracting clusters as a set of connected components from the HDBSCAN* hierarchy with each component consisting of at least m_{clSize} objects. The parameter m_{clSize} is application dependent and typically specified with the complete dataset in mind. Since the clusters are extracted based on the hierarchy built on the sampled

objects and the sample set is relatively smaller compared to the complete dataset, using m_{clSize} on a smaller dataset would fail to capture the clusters. Altering the m_{clSize} based on the sampling ratio will not capture the clusters on sampled objects since the density of the sampled objects is not known before the cluster extraction.

Due to the drawbacks of building a HDBSCAN* hierarchy based on sampled objects, we consider data summarization techniques that reflect the structure of the data better than the sampled objects.

6.2 Data Summarization Techniques

The well-known hierarchical clustering algorithms are data dependent since the distance measure between every pair of data objects is to be computed. The simplest way of decreasing run time of the algorithm is to decrease the size of the dataset on which the expensive data mining algorithm runs. This can be achieved by sampling the larger datasets and running the complete algorithm on the set of sampled objects.

Consider a dataset X with n d-dimensional data objects $\{x_i\}$, $1 \leq i \leq n$ and a set X_s of s randomly sampled d-dimensional data objects where $s << n$. If the runtime complexity of a clustering algorithm on X is $\mathcal{O}(n^2)$, then the execution time of the same algorithm on X_s (only on sampled objects) is considerably reduced to order of $\mathcal{O}(s^2)$. With fewer number of data objects, the quality of the clustering result can be considerably compromised. A trade-off between clustering quality and execution time is required.

Data summarization techniques were introduced to scale-up clustering algorithms by applying a summarization technique on the data before applying the clustering algorithm, reducing the size of the dataset and thus the overall execution time, but also trying to maintain more information about the whole dataset than a sample of the same size would contain. For partitional clustering algorithms, sufficient statistics have been proposed in [8]. BIRCH introduces the sufficient statistics, called *"Clustering Features" (CF)*. BIRCH computes compact descriptions of a sub-cluster c using a CF which is defined as

$$CF_c = (n_c, LS_c, SS_c)$$

where, n_c = number of objects in c

LS_c = Linear Sum $\left(\sum_{x_i \in c} \vec{x_i}\right)$

SS_c = Sum of Squares $\left(\sum_{x_i \in c} \vec{x_i}^2\right)$

that are included in the sub-cluster c.

More details about BIRCH have been discussed in section 2.1.2.

Based on the definition of sufficient statistics, a method of data compression was proposed in [34] for scaling up the k-means algorithm. It uses the sufficient statistics of BIRCH by summarizing different sets of data objects or their summaries independently. These sets can be categorized into

- Points that are unlikely to change their cluster membership in different iterations of the clustering algorithm.
- Data summaries that represent sub-clusters of data objects that are considered to be tight.
- Set of data objects which show anomalous behaviour and cannot be assigned to any of the other data summarizations.

Squashing [35] is another approach where the dimensions of the data space are partitioned and grouped into different regions. A set of moments like mean, minimum, maximum and second order momentums like X_i^2 and $X_i X_j$ are calculated and stored. The higher the order of momentums, the higher is the degree of approximation. The moments of the squashed items approximate those of the actual data set for each region that was defined.

Using Clustering Features or Random Sampling with hierarchical clustering algorithms, such as OPTICS [28], the clustering results suffered from two major problems, *"Structural distortion"* and *"Size distortion"* [36]. *Structural*

distortion occurs when the structure of the clusters after applying a clustering algorithm on clustering features or random sampling do not represent the structure of the database (visualized using a reachability-plot). *Size distortion* refers to the distortion in sizes of clusters, where the reachability-plots are stretched and squeezed. Also, at very high compression rates, the distance between the representative objects or the sampled objects do not reflect an approximate distance between components of the original database. As a solution to these distortion problems, Data Bubbles were introduced in [37] and applied on
OPTICS algorithm [36].

Data Bubbles overcome these drawbacks by estimating the true reachability distance between sets of data objects, thus providing sufficient statistics suitable for hierarchical clustering. Data Bubbles will be described in the next section in more detail.

6.3 Data Bubbles

Definition of Data Bubble: Let a dataset X be defined as a set of n d-dimensional data objects $X = \{x_i\}$, $1 \leq i \leq n$. Then a *Data Bubble* with respect to X is defined as a tuple

$$B_X = (rep, n, extent, nnDist)$$

- *rep* is a representative for the set of data objects in X (the representative object need not necessarily be an object of X).
- n is the number of data objects in X.
- *extent* is a real number such that most objects of X are located within a distance of extent around the representative. It is also called *"radius"* of the Data Bubble.
- *nnDist* is a function denoting the estimated average k-nearest neighbor distances within the set of objects X for values of $k, k = 1, ..., m_{pts}$.

Mathematically, the properties of the Data Bubble are computed by the following functions:

- Representative Object

$$\sum_{i=1,\ldots,n} X_i/n$$

- Extent

$$\sqrt{\frac{\sum\limits_{i=1,\ldots,n}\sum\limits_{j=1,\ldots,n}(X_i - X_j)^2}{n(n-1)}}$$

The extent can be computed from the sufficient statistics of the clustering feature of BIRCH, by the following expression:

$$extent = \sqrt{\frac{2 \times n \times SS - 2 \times LS^2}{n(n-1)}}$$

- Assuming that the distribution of data objects within the Data Bubble is uniform, Nearest neighbor estimate is given by

$$nnDist(k, B_X) = \left(\frac{k}{n}\right)^{\frac{1}{d}} \times extent$$

Initializing Data Bubbles: Given a dataset X with n d-dimensional data objects, m Data Bubbles can be constructed using X by applying the following steps:

1. Draw a random sample of m data objects from the n objects in X. This set of m objects is called the *"seed set"*, S. Each object in S is called *"seed object"*.

2. Initialize a set of m Data Bubbles with each Data Bubble containing one seed object from S.

3. For each object in $X - S$, find the nearest *seed object* in S and update the sufficient statistics, such as n, LS and SS, from which a Data Bubble tuple can be calculated.

Using the definition of Data Bubbles and its properties, such as extent, representative and nearest neighbor estimate, other metrics can be defined for Data Bubbles.

Distance between two Data Bubbles: Consider two Data Bubbles, $B = (rep_B,\ n_B,\ extent_B,\ nnDist_B)$ and $C = (rep_C,\ n_C,\ extent_C,\ nnDist_C)$, then the distance between two Data Bubbles B and C can be defined as

$$dist(B,C) = \begin{cases} 0 & if B = C \\ dist(rep_B, rep_C) - (extent_B + extent_C) & if dist(rep_B, rep_C) - \\ \quad +nnDist(1,B) + nnDist(1,C) & \quad (e_1 + e_2) \geq 0 \\ max(nnDist(1,B), nnDist(1,C)) & otherwise \end{cases}$$

This means that the distance between two Data Bubbles is equal to zero if the Data Bubbles are same ($B = C$). If two Data Bubbles do not overlap with each other, i.e., when the distance between the representatives of the Data Bubbles exceed the sum of their radii, the distance between two Data Bubbles is given by the distance of their centers minus their radii plus their expected nearest neighbor distances. When two Data Bubbles overlap, then the distance between two Data Bubbles is the maximum of their estimated neighbor distances. The distance between two data Bubbles is illustrated using figures 6.3 and 6.4.

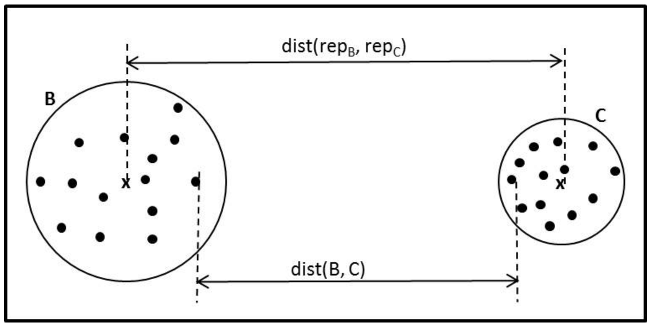

Figure 6.3: Distance between two Data Bubbles - Non-overlapping Data Bubbles

Core Distance of a Data Bubble: We adapt the definition of core distance of a Data Bubble from [36] to suit the definition of *core distance* of the

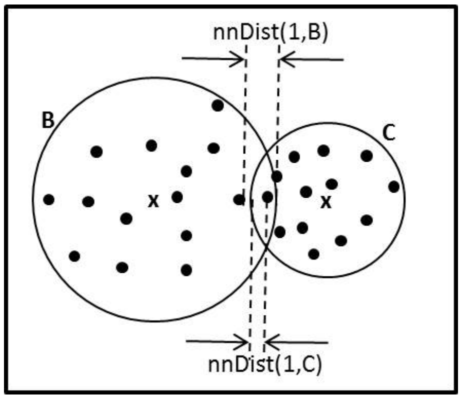

Figure 6.4: Distance between two Data Bubbles - Overlapping Data Bubbles

HDBSCAN* algorithm [7]. The core distance of a Data Bubble defined for OPTICS, is designed w.r.t. ϵ-neighborhood and m_{pts} while the m_{pts}-neighborhood in HDBSCAN* is not restricted by the parameter ϵ.

Let N be a set of m Data Bubbles for a given dataset X with n data objects. Consider a Data Bubble $B = (rep_B,\ n_B,\ extent_B,\ nnDist_B)$, then the core distance of a Data Bubble B is given by

$$core\text{-}dist_{m_{pts}}(B) = d_{core}(B) = dist(B, C) + nnDist(k, C)$$

where C is a Data Bubble in the set of Data Bubbles, N, such that the distance between B and the closest Data Bubble C (along with all the Data Bubbles which are closer to B than C) contain together at least m_{pts} objects.

$$\sum_{\substack{X \in N \\ dist(B,X) \leq dist(B,C)}} n \quad \leq \quad m_{pts}$$

and the value of k is given by

$$k \quad = \quad m_{pts} - \sum_{\substack{X \in N \\ dist(B,X) \leq dist(B,C)}} n$$

If the Data Bubble B, contains more than m_{pts} data objects in it, then $C = B$ and the core distance is given by $nnDist(k, C)$, i.e., the definition states that if the Data Bubble represents at least m_{pts} objects, then core distance is the estimated m_{pts} neighbor distance of that Data Bubble. If the Data Bubble B contains less than m_{pts} data objects in it, find a Data Bubble C such that all Data Bubbles that are nearer to B than C (Data Bubbles P and Q in figure 6.5) together with Data Bubble B have sum of the number of data objects contained in them less than m_{pts}, and the core distance is given by the sum of the distance between Data Bubbles B and C and the k-nearest neighbor estimate in C, with k defined as above.

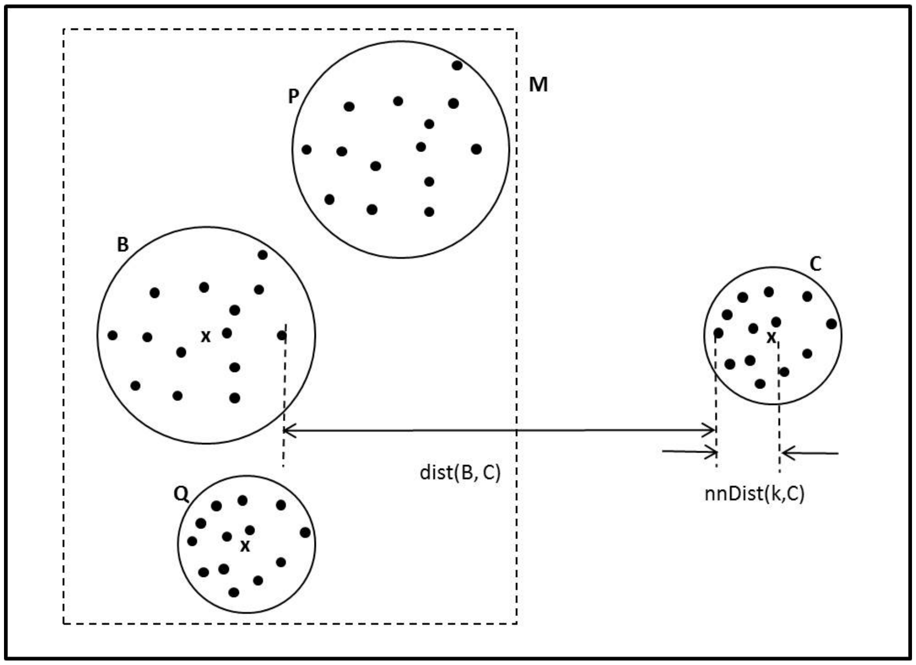

Figure 6.5: Core Distance of a Bubble when $n_B < m_{pts}$

Mutual Reachability Distance: The HDBSCAN* hierarchy is created on the basis of constructing a Minimum Spanning Tree on the mutual reachability graph. Therefore we introduce the definition of the *mutual reachability distance* between a pair of Data Bubbles. The *mutual reachability distance* between two Data Bubbles, B and C, with respect to m_{pts}, is given by

$$d_{m_{reach}}(B, C) = max(d_{core}(B), d_{core}(C), dist(B, C))$$

6.4 Implementation of Data Bubbles in PHDBSCAN

This section explains the implementation of PHDBSCAN using Data Bubbles. The algorithm has two major steps, *"Recurse"* and *"Combine"* similar to that explained in Algorithm 6 in section 5.2. Data Bubbles are implemented in the Divide method of the *Recurse* step which is shown in Algorithm 12.
Figure 6.6 illustrates the flow of the *Recursive Sampling Approach* using Data Bubbles. Each data block is divided into clusters which are further refined by sending the clusters to a processing unit, dividing the data recursively until the *leaf MSTs* are formed.

We will only discuss the procedures that are different from the implementation of PHDBSCAN on sampled objects explained in Chapter 5.

Exclusion of Noise objects: For a given dataset Y, data objects in Y that belong to Data Bubbles in set O (set of Data Bubbles considered noise) can be discarded as noise objects. All data objects that belong to Data Bubbles in set O are indeed noise objects. This is proved by considering the following two scenarios:

1. **Size of a Data Bubble is greater than m_{clSize}:** Any given Data Bubble, $B_p = (rep_p, n_p, extent_p, nnDist_p)$, with $n_p \geq m_{clSize}$ is always classified as a cluster. Therefore, no Data Bubble with $n_p \geq m_{clSize}$ is added to O.

2. **Size of a Data Bubble is less than m_{clSize}:** All Data Bubbles which are added to set O are Data Bubbles with $n_p < m_{clSize}$. While extracting clusters, any Data Bubble (or set of connected Data Bubbles) formed due to the removal of an edge or edges with weight ϵ, are considered *noise*. Since a Data Bubble represents all the data objects included in it and the property of the Data Bubble n_p is used to calculate if a set of connected Data Bubbles can form a cluster, any Data Bubble (and the data objects it comprises) in set O can be safely assumed to be noise.

```
Algorithm 12: $PHDBSCAN_{bubbles}$ Divide
  Input: Dataset $Y$, Parameter $m_{pts}$, Division Method $div$
  Output: Set of Divided Components: $\{Y_1, ..., Y_k\}$, Set of
          Inter-connecting Edges $E_{inter}$
    1. Draw a sample $S$ of size $m$ from $Y$. The unsampled objects belong
       to the set $US = Y - S$.
    2. Using $S$ as the seed set of size $m$, initialize set of $m$ Data Bubbles
       $B = \{B_1, ..., B_m\}$.
    3. For each object in $US$, find the nearest neighbor in $S$ and update the
       corresponding Data Bubble in $B$.
    4. With a set of Data Bubbles $B$, construct a HDBSCAN* hierarchy:
       (a) Compute an MST, $MST_{bubbles}$ on the Mutual Reachability Graph,
           $G_{m_{reach}}(B)$ computed on Data Bubbles.
       (b) Construct HDBSCAN* hierarchy based on $MST_{bubbles}$.
       (c) Extract prominent clusters from the constructed HDBSCAN*
           hierarchy, $C_S = \{C_{s_i}\}$, $1 \leq i \leq k$ along with a set of "noise"
           objects, $O$. Clusters are extracted using different approaches
           based on the Division Method $div$.
           if $div = Binary$
              Call Algorithm 9 BinaryDivision (HDBSCAN*
              hierarchy)
           else
              Extract the most prominent clusters from
              HDBSCAN* hierarchy using the concept of Excess of Mass
              as shown in Algorithm 3.
       Cluster extraction returns a list of clusters sets $C = \{C_1, ..C_k\}$ and a
       set of Data Bubbles considered to be noise $O$. Each set consists of all
       the Data Bubbles classified to one of clusters in $C$ or the set $O$.
    5. For each object in $Y$, find the cluster membership of Data Bubbles
       among the set $\{C_1, ..., C_k\}$ to which the object was mapped in Step 2
       and assign the data object to that cluster. Objects that belong to
       Data Bubbles in $O$ are considered noise and are discarded.
    6. Add the edges in $MST_{bubbles}$ that connect Data Bubbles in different
       clusters to $E_{inter}$.
```

Scenario where noise objects w.r.t. complete dataset are not eliminated: Although all noise objects identified can be safely removed, not all

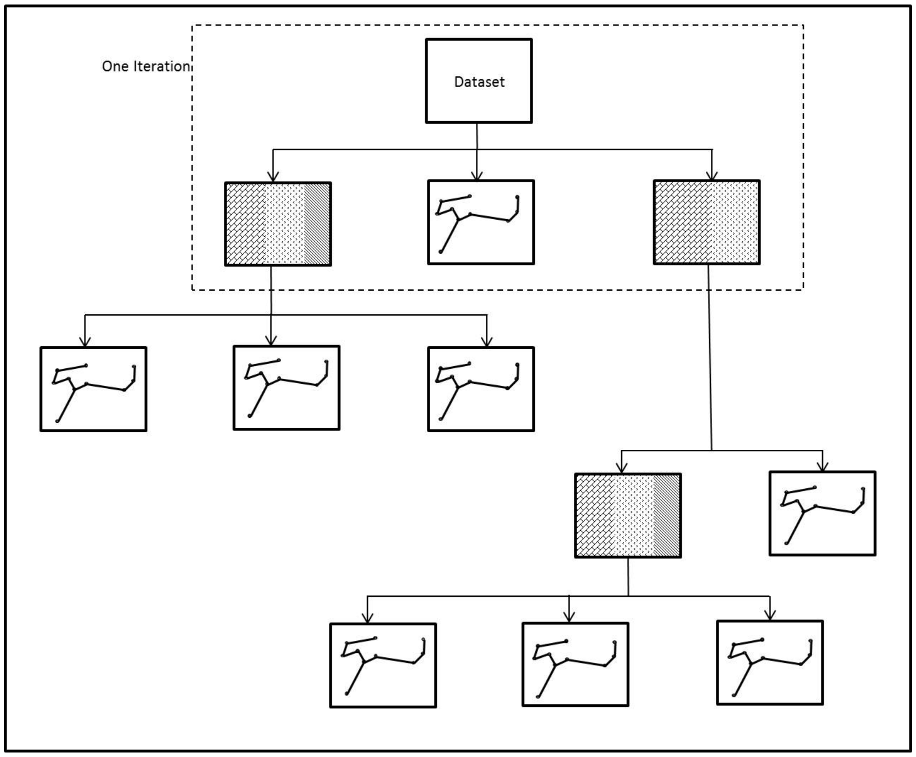

Figure 6.6: Recursive Sampling Approach with Data Bubbles

noise objects are identified immediately using Data Bubbles in the PHDBSCAN algorithm. Consider the dataset shown in figure 6.7, with seed objects denoted by x,y and z for Data Bubbles A, B and C respectively. The Data Bubble C is identified as noise and eliminated from the hierarchy. For the same dataset when different seed objects have been selected, as shown in figure 6.8, the data objects which were removed as noise in the previous scenario are now a part of a Data Bubble C which cannot be eliminated and which is recursively sent for further processing.

Finding Inter-Cluster Edges: The *Inter-cluster edges* are identified in a similar way as finding the inter-cluster edges using sampled objects (explained in section 5.2). The shortest edges between the extracted prominent clusters are added to the set E_{inter}. Edges in E_{inter} have weights corresponding to the shortest mutual reachability distance between Data Bubbles in different clusters. These identified edges give a better estimate of the shortest distance between clusters than the estimate computed using only sampled objects.

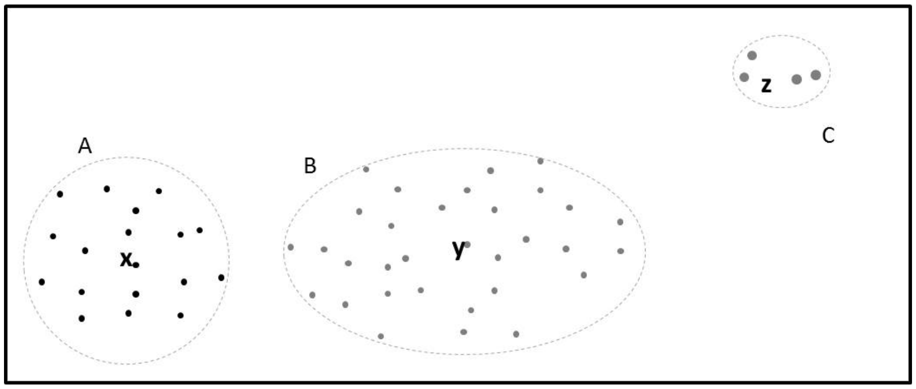

Figure 6.7: Example where noise would be excluded from further processing

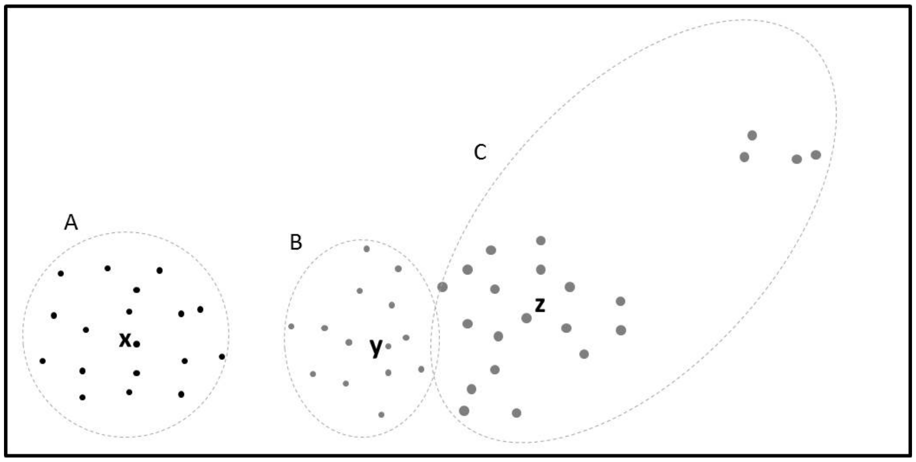

Figure 6.8: Example of a scenario where noise is included for further processing

Dividing Data and sending it to different Processing Units: Data objects which belong to the same cluster form a data block. Different data blocks are sent to different processing units for further refinement.

6.4.1 Multi-Node Cluster Extraction

In the N-ary method of Recursive Sampling Approach using Data Bubbles, at each level of recursion, PHDBSCAN algorithm finds a number of prominent clusters by constructing a HDBSCAN* hierarchy and extracting the most prominent clusters as flat non-overlapping partitions. The cluster stabilities used to find the prominent clusters are based on the Data Bubbles which gives an estimate of the stabilities of clusters with respect to the complete dataset. Using these stabilities, it is possible to construct a *"Distributed Cluster Tree"* by identifying cluster stabilities at distributed nodes across all levels of recursion. This *Distributed Cluster Tree* can be used to extract the prominent clusters with respect to the complete dataset without combining the local MSTs to form a combined MST and computing the HDBSCAN* hierarchy from a combined MST.

Consider the dataset shown in figure 5.3 in chapter 5. Using the N-ary approach, the *Distributed Cluster Tree* (illustrated in figure 6.9) is identified. In figure 6.9, leaf clusters are shown using shaded squares and the data blocks within the same processing units are shown using rectangles with broken lines. The stability of each cluster extracted at the processing unit is computed immediately. The parent-child relationship is maintained by keeping track of the clusters sent for further processing. The parent cluster is the component that is sent for further refinement and children clusters are clusters identified in the next immediate level of refinement.

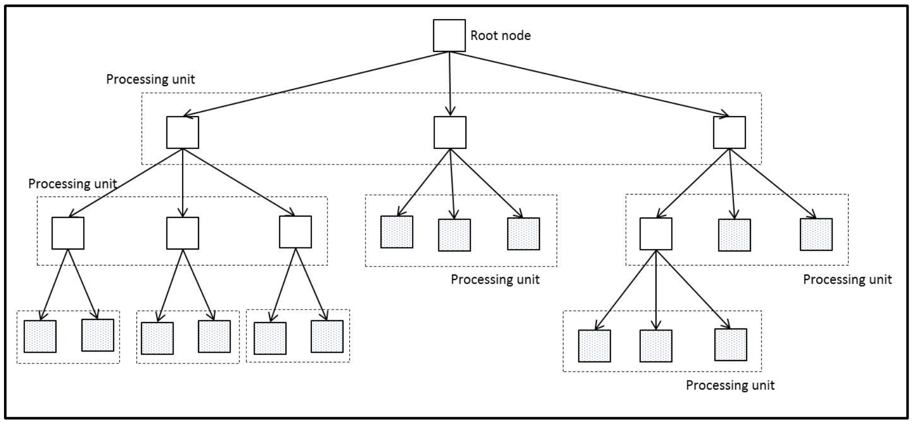

Figure 6.9: Distributed Cluster Tree for data shown in figure 5.3

The flat non-overlapping partitioning from a *distributed cluster* tree can be seen as an optimization problem similar to the one given in equation 4.1 and solved using Algorithm 3. It can be extracted by solving an optimization problem with the objective of maximizing the overall stabilities of the extracted clusters. We process every node except the root, starting from the leaves (bottom-up), deciding at each node C_i whether C_i or the best-so-far selection of clusters in C_i's subtrees should be selected. To be able to make this decision locally at C_i, we propagate and update the total stability $\hat{S}(C_i)$ of clusters selected in the subtree in the following recursive way:

$$\hat{S}(C_i) = \begin{cases} S(C_i), & if\ C_i \in leaf\ clusters \\ max\left\{ S(C_i), \sum\limits_{C_{i_c} \in \{C_{i_c}\}} \hat{S}(C_{i_c}) \right\} & if\ C_i \notin leaf\ clusters \end{cases}$$

where set C_{i_c} is the set of child clusters of node C_i and $\hat{S}(\cdot)$ gives the stability of the respective clusters.

6.4.2 Shortcomings of using Binary method of cluster extraction

There are a couple of shortcomings of using the *Binary* extraction method in the *Recursive Sampling Approach.*

1. The *Binary* division method has a drawback with respect to the sizes of the clusters that are formed. If the data is distributed in such a way that the first formed clusters almost always end up getting divided into two clusters, a very large and a relatively smaller cluster, then the number of recursion steps becomes very large. With a larger depth of recursion, the amount of resources needed to maintain the recursion stack also increases. The overhead for maintaining the additional resources may not be negligible causing a serious overhead on computation and time. The binary approach works well for the data that is distributed in such a way that the first 2 clusters that are extracted are indeed the most prominent clusters for the complete dataset.

2. The other drawback of using Binary division method is that the *"Multi-node cluster extraction"* cannot be implemented. *Multi-node cluster extraction* depends on the cluster stabilities estimated at distributed nodes to construct the *Distributed Cluster Tree.* The cluster stabilities cannot be computed in the Binary division method since the data is immediately sent for further processing as soon as the first two prominent clusters are identified from the HDBSCAN* hierarchy. The hierarchy is not traversed any further and hence computing the cluster stabilities is not possible,

making *multi-node cluster extraction* not viable for the *Binary* division method.

Chapter 7

Implementation using the Map Reduce Framework

7.1 MapReduce Framework

MapReduce is a programming framework [38] to process large scale data in a massively data parallel way. MapReduce Framework has several advantages over the other existing parallel processing frameworks. The programmer is not required to know the details related to data distribution, storage, replication and load balancing, thus hiding the implementation details and allowing the programmers to develop applications/algorithms that focus more on processing strategies. The programmer has to specify only two functions, a *map* and a *reduce* function. The framework (explained in [39]) is summarized as follows:

1. The map stage passes over the input file and output (key, value) pairs.
2. The shuffling stage transfers the mappers output to the reducers based on the key.
3. The reduce stage processes the received pairs and outputs the final result.

Due to its scalability and simplicity, MapReduce has become a promising tool for large scale data analysis. The MapReduce framework interleaves sequential and parallel computation. MapReduce consists of several rounds of computation. For a given set of distributed systems (nodes), without any communication between nodes, each round distributes the data among the set of available nodes. After the data is distributed, each node performs the required computation on the data available to them. The output of these computations is either combined to form the final result or sent to another round of computation depending on the application's requirement. In the MapReduce programming

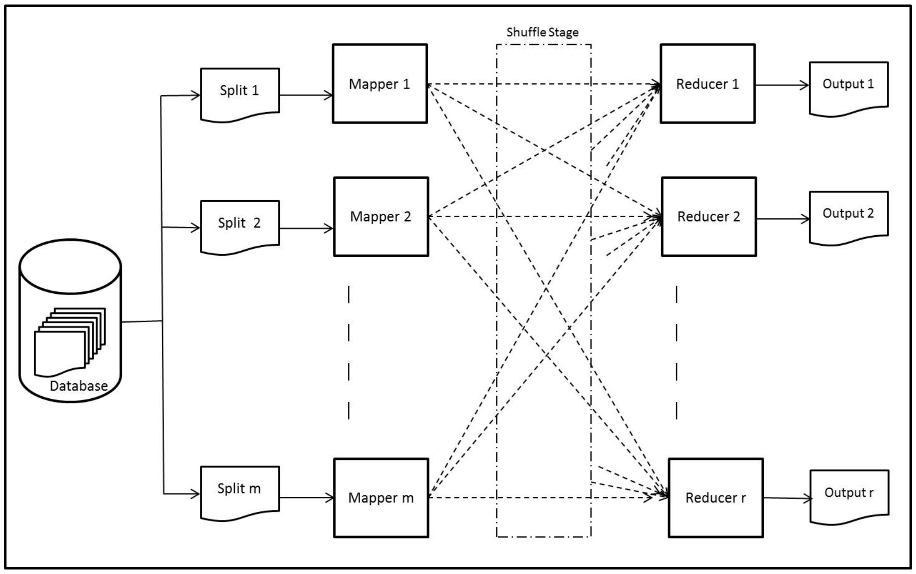

Figure 7.1: An example of a Map Reduce Framework

paradigm, the basic unit of information is a $(key, value)$ pair where both key and $value$ are binary strings. The input to any MapReduce algorithm is a set of $(key, value)$ pairs. Operations on a set of $(key, value)$ pairs occur in three different stages, *Map*, *Shuffle* and *Reduce*. They are explained as follows:

Map Stage: In the map stage (illustrated in figure 7.2), the map function of the mapper takes a single $(key, value)$ pair as input, and produces an output in the form of $(key, value)$ pairs. The number of outputs produced by the mapper is the number of times the map function is called, i.e., the number of $(key, value)$ pairs that were provided to the mapper. The reason that the map operation operates on only one pair of $(key, value)$ at a time is to make it stateless, allowing the process of parallelization to be easy and data independent on different nodes [39].

Shuffle Stage: During the shuffle stage, the system that implements MapReduce sends all values associated with an individual key to the same machine.

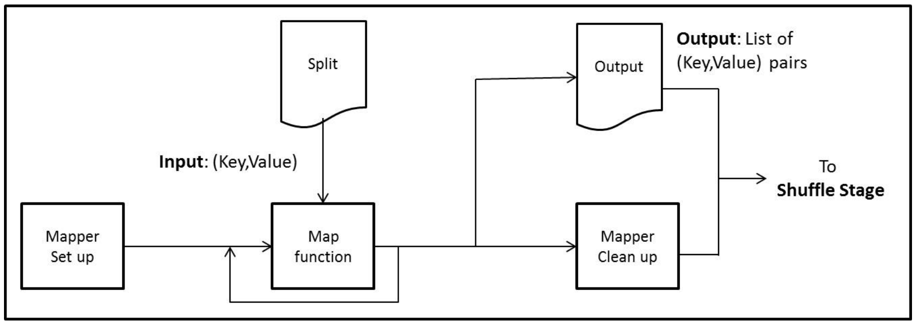

Figure 7.2: Flow of a Mapper in a MapReduce Framework

This is an inbuilt scheme of MapReduce and the programmer does not have to be aware of the implementation of distribution of data with the same key to a single node.

Reduce Stage: During the reduce stage, the reducer takes all values associated with a single key k, and outputs a set of $(key, value)$ pairs. This portion of computation is the sequential aspect of MapReduce. Successful completion of all mappers is the only criterion for a reducer to start its computation. Any sequential steps for the subset of data that are mandated by the algorithm or the process could be implemented in the reduce stage. The flow within a reducer in a MapReduce framework is illustrated in figure 7.3.

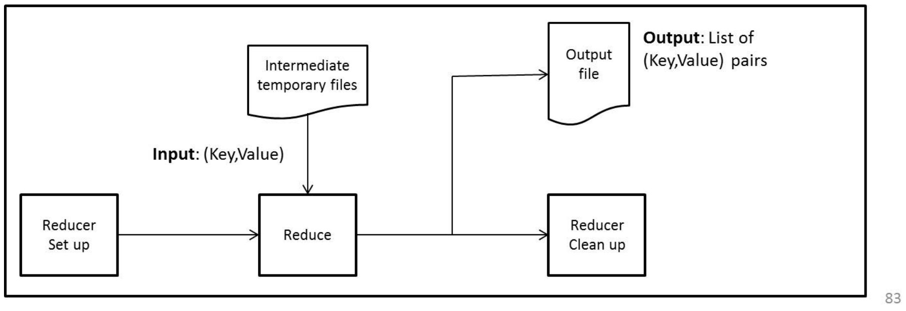

Figure 7.3: Flow of a Reducer in a MapReduce Framework

7.2 Implementation of Random Blocks Approach

The implementation of *Random Blocks Approach* is similar to that explained in [1] without the use of special data structures. The algorithms for the implementation of *Random Blocks Approach* is shown in Algorithm 13, 14 and 15. **Master Processing Node:** The master processing node is responsible for generating the data blocks and calling the mappers and reducers. Each data block is considered a *“data split”*. A *“data split”* or *“split”* is a file or part of a file that is sent to a mapper for independent processing. The size of a *“split”* is called *“split size”*. Data blocks are created in such a way that the size of a data block should not exceed the maximum split size allowed by the implementation of MapReduce. If the *split size* exceeds, then the split is divided into smaller data splits on which the true reachability distances between all pairs of data objects cannot be computed in at least one of the several available data blocks. The flow is illustrated in figure 7.4.

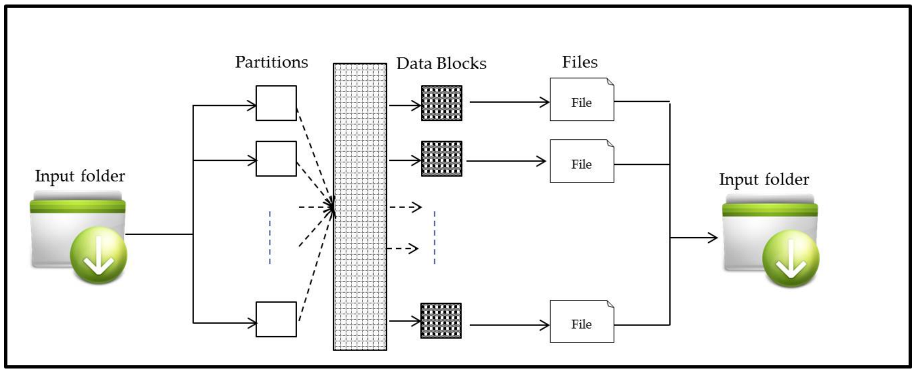

Figure 7.4: Flow of events at the Master Processing Node in the Random Blocks Approach

Mapper: The original dataset is divided into k partitions that are combined to form l *“data blocks”* as explained in section 5.1. Each data block is sent to a mapper. Each mapper consists of a map function which takes the input in the form of a $(key, value)$ pair. The number of times a map function is called within a mapper is the number of data objects available within a data block. Each data block consists of a unique identifier which serves as the *key* and the data object serves as the *value*. A data object consists of a data object identifier, *DataObjectID* and its feature vector. Once all the data objects with the same key (data objects within the same block) are aggregated by the map function, compute the core distances of all the objects w.r.t m_{pts}. Compute a mutual reachability graph, $G_{m_{reach}}$ on which the local Minimum Spanning Tree is computed. The output of this local MST is stored in the file system in the form of a $(key, value)$ pair. Since all the local minimum spanning trees are sent to one single reducer, a common value for *key* is used (say, 1) and the edge given by $(u, v, d_{m_{reach}}(u, v))$ is the corresponding *value*. Figure 7.5 illustrates the Mapper in the Random Blocks Approach.

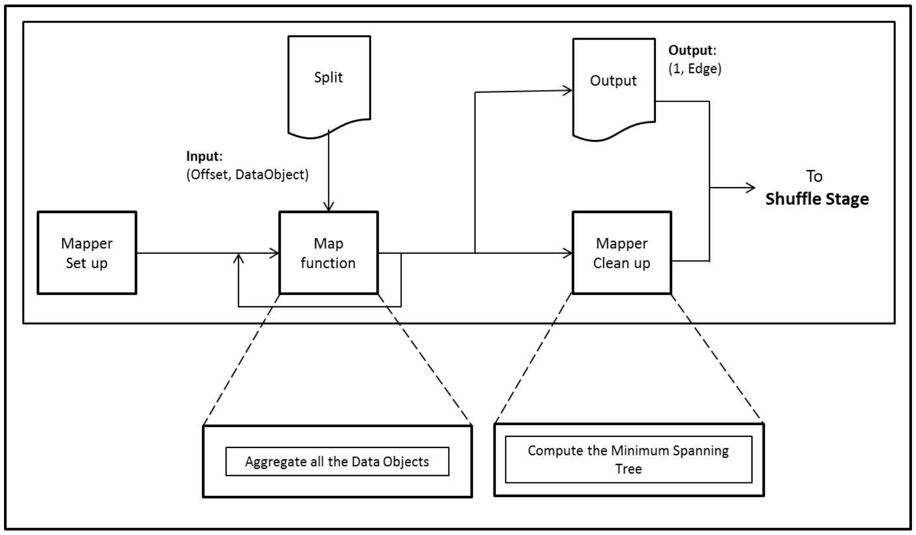

Figure 7.5: Mapper in the Random Blocks Approach

Reducer: The single reducer takes in all the local Minimum Spanning Trees from all the nodes. Reducer implements its own version of Spanning Tree algorithm to prune larger edges. Kruskal implementation of the Minimum Spanning Tree algorithm is more efficient for sparse graphs. Algorithm 15 explains the steps at the reducer. The final output of the reducer is a combined MST which is written back to the file system in the form of $(key, value)$ pairs. Since the reducer returns to the master node, the *key* is *null* and each edge of the combined MST is the *value*. Figure 7.6 illustrates the Reducer in the Random Blocks Approach.

A simple architecture to implement this method is shown in the figure 7.7.

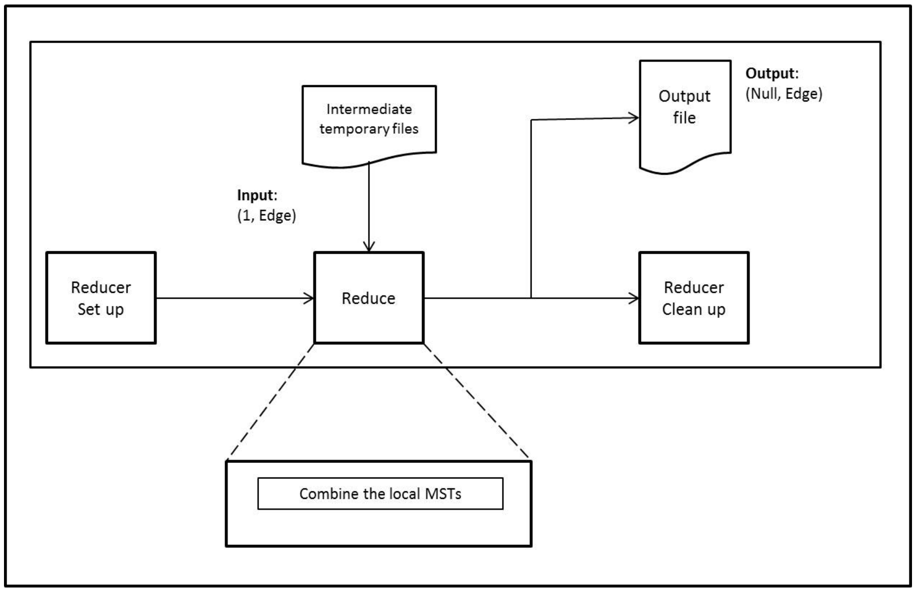

Figure 7.6: Reducer in the Random Blocks Approach

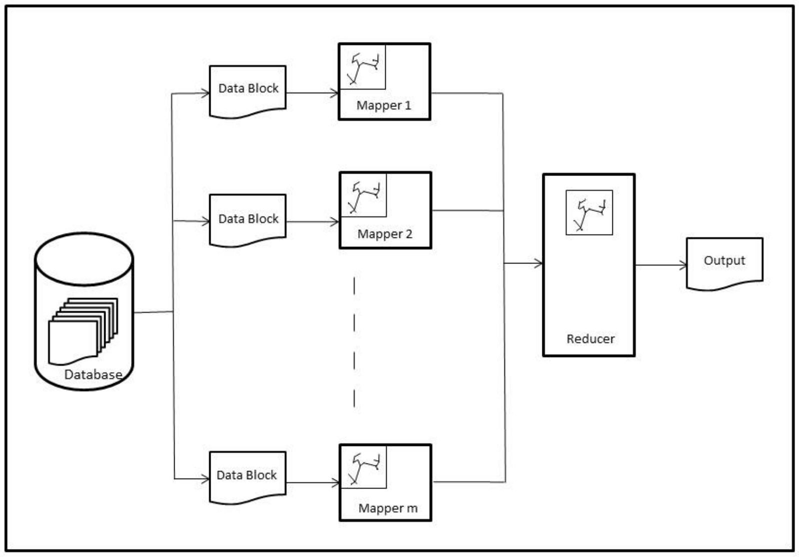

Figure 7.7: Simple flow of Random Block Approach in MapReduce Framework

7.3 Implementation of PHDBSCAN

The recursive model of the PHDBSCAN algorithm can be converted into an iterative algorithm to fit the MapReduce framework. At the time, the experiments were designed, the MapReduce framework did not include a setup to support recursion.

7.3.1 PHDBSCAN Implementation using only Sampled Objects

Recursive Sampling Approach using only sampled objects is the first method implemented using MapReduce Framework. Consider a whole dataset Y is available in a file or set of files in a directory called *"input directory"*. The *master processing node* draws a random sample and HDBSCAN* hierarchy is built based on these sampled objects followed by the extraction of the most prominent clusters. The data objects along with their cluster label is written to a file called *"Hierarchy file"*.

The master processing node calls the mappers with the *input directory.* The MapReduce framework is responsible to send all the files to different distributed nodes. Each file is sent to a different node if the size of the file is less than the *"split size"*. Maximum allowed *split size* per *split* depends on the version of implementation of the MapReduce framework (e.g., Hadoop). If the size of any file in the input directory is more than the *split size*, the file is divided into two or more data splits and sent to different mapper nodes. Each data split corresponds to a *data block.*

The hierarchy file is sent to all mappers by the master processing node. Sending the same file to all the distributed nodes can be achieved by using the *"distributed cache"* mechanism in MapReduce framework. The framework sends the hierarchy file to all the nodes assigned to be the mappers.

Master Processing Node: The Master Processing Node, also called *"Driver Node"* or *"Driver"*, is the starting point of the algorithm. The *input* and *output directory* are given as input arguments with all the other parameters (m_{pts}, m_{clSize}, τ, div) stored in the configuration file that is accessible by all the nodes. A single call to a set of mappers and reducers is called a *"job"*. Each job has its own *input* and *output directory.*

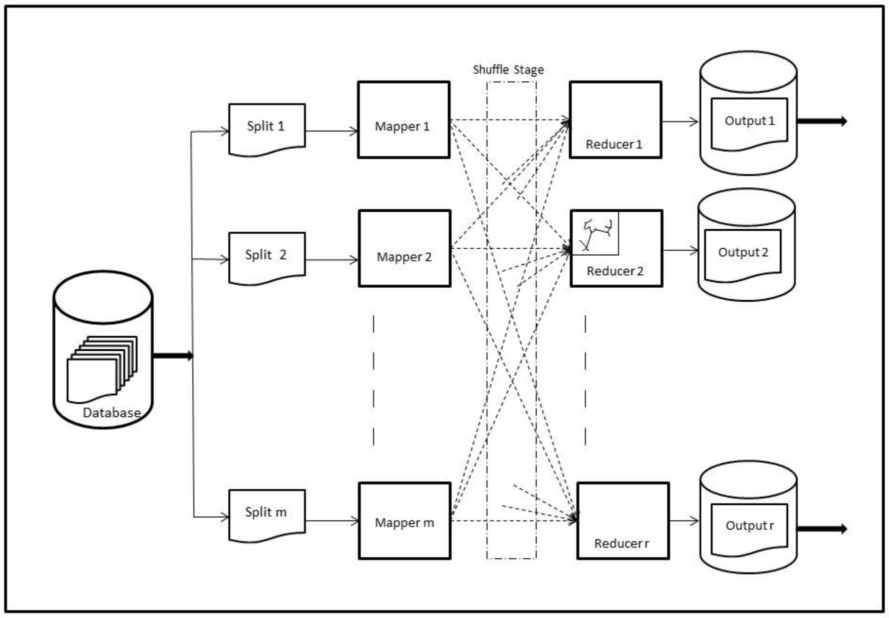

Figure 7.8: Flow of Recursive Sampling without Bubbles in MapReduce Framework

At *Driver*, a random sample for the complete dataset is drawn and a *"Hierarchy file"* is created. A job is created with *input* and *output directories* as input arguments along with the hierarchy file in *distributed cache.* The job then makes a call to mappers and reducers.

Recursive Sampling Mapper: A mapper receives the data split and the hierarchy file. A mapper in this approach is used to classify all the data objects to one of the clusters extracted using HDBSCAN* hierarchy on sampled objects. Within each mapper, the *map* function is called with a $(key, value)$ pair. The *key* is the data split identifier (which is assigned by the framework) and the *value* is a data object, x_p. A map function is used to identify the cluster membership of the nearest neighbor of x_p in the set of sampled objects (available in the hierarchy file), and assign the cluster label to x_p. The output of map function is also a $(key, value)$ pair, where *key* is a cluster label and *value* is a data object.

Shuffle Stage: The Shuffle stage is an inbuilt mechanism of MapReduce

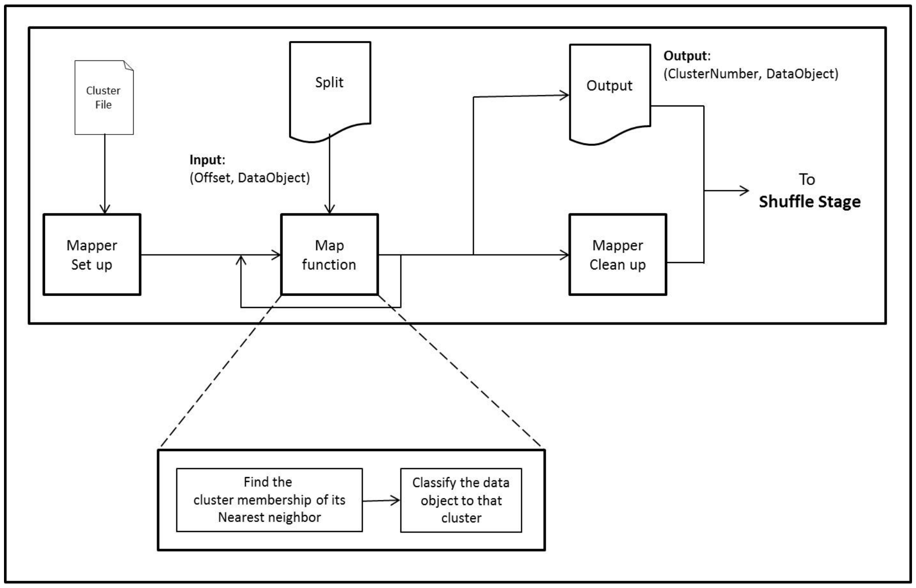
Cluster File
Split
Input:
(Offset, DataObject)
Output
Output:
(ClusterNumber, DataObject)
To
Shuffle Stage
Mapper Set up
Map function
Mapper Clean up
Find the cluster membership of its Nearest neighbor
Classify the data object to that cluster

Figure 7.9: Mapper in the Recursive Sampling Approach

framework and is not required to be explicitly written by the programmer. The Shuffle stage gathers the output from all the map functions of all the mappers and redirects the data (stored in *value* of the output $(key, value)$ pairs) associated with an individual *key* to the same node, called reducer. All the data objects that are classified to a single cluster (by having the same cluster label) are sent to the same reducer. The number of reducers called will correspond to the number of clusters that were formed using the sampled objects.

Recursive Sampling Reducer: The reducer implements the sequential aspect of the MapReduce framework. Reducer starts immediately after all the mappers are terminated successfully. Each reducer checks for the number of data objects (corresponding to the number of values) received by it. If the number of data objects are greater than the processing capacity of the node, given by τ, a random sample is drawn from the set of available data objects. HDBSCAN* hierarchy is computed on sampled objects w.r.t m_{pts}, prominent clusters are extracted on the hierarchy and data objects along with their cluster membership are written to a *"Hierarchy file"*. The *inter-cluster edges* are also computed and written to a different directory called *"Inter Cluster*

Edges". The optional parameter of ς to check the sampling tolerance can be used here, once the inter-cluster edges are identified. If the sampling tolerance is not satisfied, the reducer can return with a fail status. All the data objects in this reducer are written to an *output directory* which will serve as an *input directory* for future iterations. Each reducer writes its output in a unique directory. If the number of data objects are less than τ, then an MST is computed based on all the data objects w.r.t m_{pts} and the output is written to a different directory called *"local MST directory"*. Reducer returns to the master processing node with a success status.

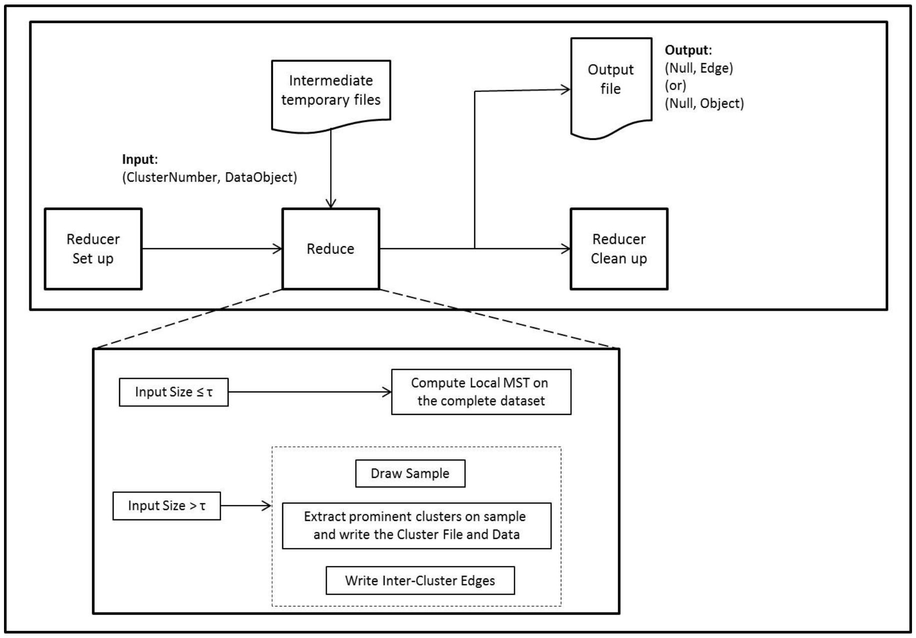
Intermediate temporary files
Output file
Output:
(Null, Edge)
(or)
(Null, Object)
Input:
(ClusterNumber, DataObject)
Reducer Set up
Reduce
Reducer Clean up
Input Size ≤ τ
Compute Local MST on the complete dataset
Draw Sample
Input Size > τ
Extract prominent clusters on sample and write the Cluster File and Data
Write Inter-Cluster Edges

Figure 7.10: Reducer in the Recursive Sampling Approach

Master Processing Node: Every time a reducer returns a *success* flag, a new job is created with the next available output directory written by a reducer. This output directory will serve as an input directory for future *jobs*. The iteration continues until all the output directories written by all the reducers across all the jobs have been processed. When all the jobs are completed, the *Driver* reads all the data available within *Inter Cluster Edges* and *local MST directory*. All the edges combined together gives the approximate combined MST of the complete dataset on which the HDBSCAN* hierarchy can be computed. The algorithms for Recursive Sampling approach with only sampled objects are explained in Algorithms 16, 17 and 18.

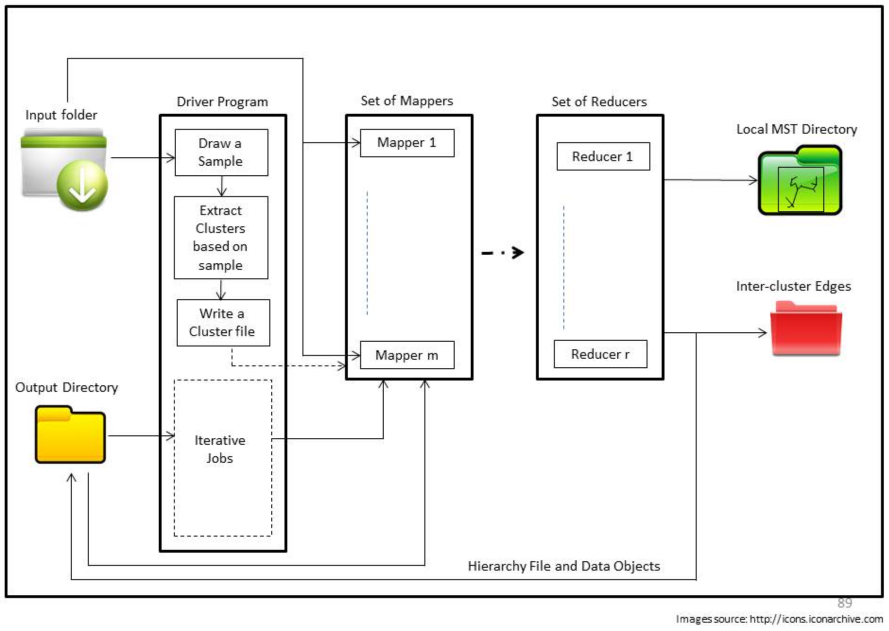

Figure 7.11: Master Processing Node in the Recursive Sampling Approach

The flow of a single iteration of the explained method is shown in figure 7.8. The whole dataset is divided into *“data splits”* by the framework where every data split is sent to one of the available mappers. Output from all the mappers are sent to the shuffle stage (shown by dotted arrows between the Mappers and the Reducers) which directs output from all the Mappers with the same key to the same reducer. The Reducer generates an output based on the threshold τ. The output can either be a subset of data with sampled objects for future iterations or *”local MSTs”* (shown with a sample MST in reducer 2 of figure 7.8). If the output is not a local MST, then the subset of data along with the *Hierarchy file* and *inter-cluster edges* are written to their respective directories for further processing.

7.3.2 PHDBSCAN Implementation using Data Bubbles

The PHDBSCAN algorithm using Data Bubbles is implemented using two layers of mappers and reducers. The first layer is used to create Data Bubbles, build HDBSCAN* hierarchy based on Data Bubbles and extract clusters based on the created hierarchy. The second layer classifies all the data objects and refines the hierarchy. The flow of such a setup is shown in figure 7.12.

Driver (Master Processing) Node: The *"Driver"* first draws a sample set called *Seed set* and writes it to a file, *"Seed file"*. This *Seed set* is sent to all the mappers of Layer 1, called *"Bubble Mappers"* through distributed cache. The complete dataset which is stored in the file system is distributed to different mappers as *data splits*. The flow of events of the Mapper are shown in figure 7.3.2.

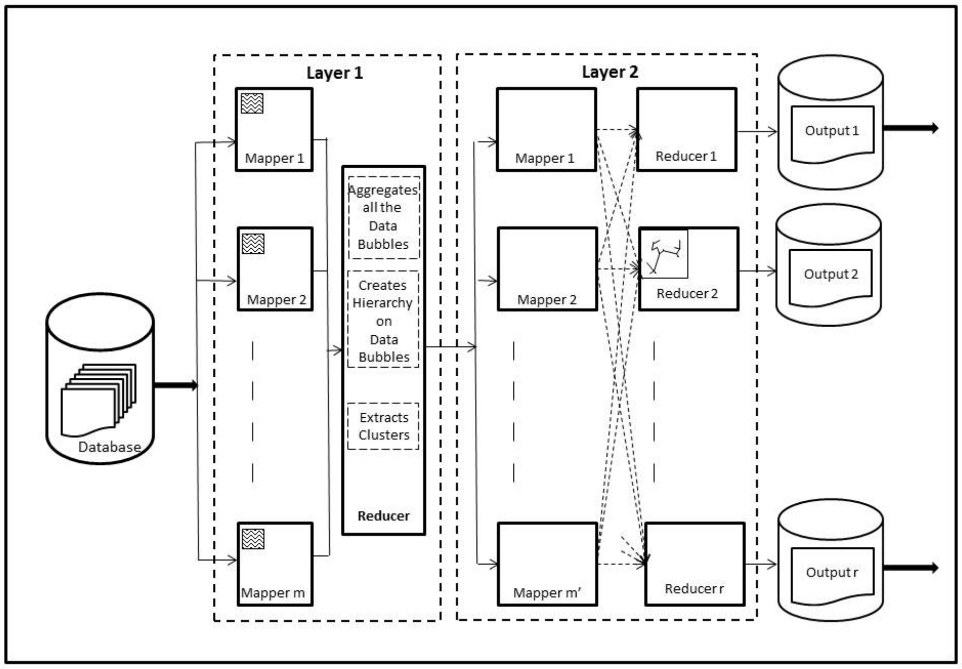

Figure 7.12: Flow of Recursive Sampling using Bubbles in MapReduce Framework

Layer 1. Bubble Mapper: *Bubble Mapper* receives a *data split* along with the *Seed file*. The seed file consists of a set of seed objects S. For each seed object in S, initialize a Data Bubble. Let the set of Data Bubbles built

on a data block l be represented as set $B(Y_l) = \{B(Y_l)_1, ..., B(Y_l)_m\}$ where Y_l is the set of data objects in data block l and m is the number of seed objects. Each map function within a mapper receives the input in the form of $(key, value)$ pair where key is a unique identifier of a *data split* and $value$ is a data object. For each object x_p received by the map function, find the nearest neighbor in S and update the corresponding Data Bubble in B_l with sufficient statistics. Update the data object received with the Data Bubble Identifier (BubbleID) to which it was mapped. For each Data Bubble in $B(Y_i)$, write the output in the form of $(key, value)$ pair with $key = 1$ and the value is the *"Data Bubble"* which consists of the BubbleID (as a one to one mapping with the seed object) and a tuple *(*n, LS, SS). An output key of 1 across all *Bubble Mappers* makes sure that all the output *values* reach a single reducer.
Layer 1. Bubble Reducer: The layer 1 consists of a single reducer which

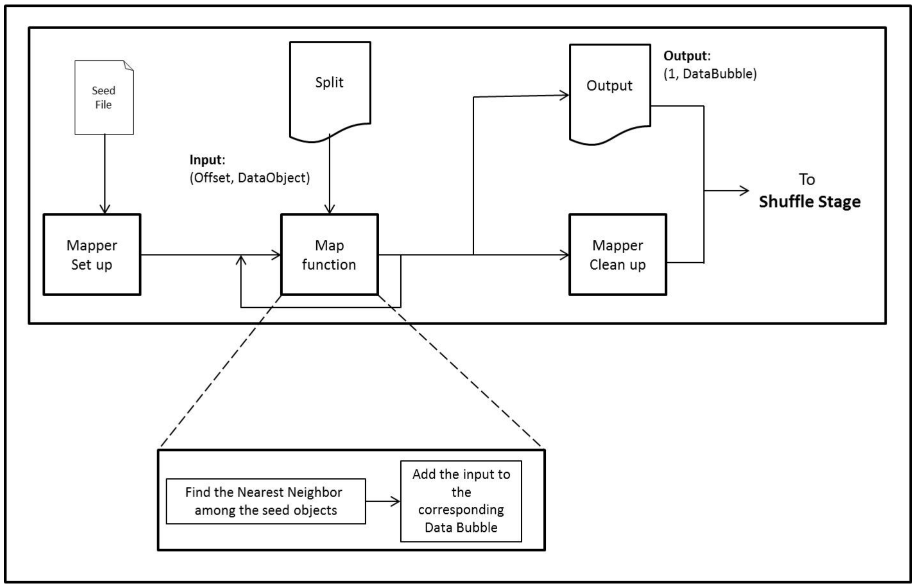

Figure 7.13: Layer 1 Mapper in the Recursive Sampling Approach using Data Bubbles

aggregates the sufficient statistics from l Bubble Mappers, where l corresponds to the number of data splits (and the number of successful mappers that were executed). Each Data Bubble set consists of m Data Bubbles and there are l such sets. Each Data Bubble in a set is aggregated with Data Bubbles across other Data Bubble sets with the same identifier is given by

$$B(Y)_i = B(Y_1)_i + ... + B(Y_l)_i;\ 1 \leq i \leq m$$

where i is a unique identifier of a Data Bubble and Data Bubbles are aggregated across l partitions.

Given two Data Bubble sets from data blocks Y_1 and Y_2, $B(Y_1) = \{B(Y_1)_1, ..., B(Y_1)_m\}$ and $B(Y_2) = \{B(Y_2)_1, ..., B(Y_2)_m\}$, each Data Bubble containing the sufficient statistics (n, LS, SS), the Data Bubbles inside these sets are individually aggregated by

$$B(Y_1)_i + B(Y_2)_i = \Big(n_{B(Y_1)_i} + n_{B(Y_2)_i}, LS_{B(Y_1)_i} + LS_{B(Y_2)_i}, SS_{B(Y_1)_i} + SS_{B(Y_2)_i}\Big)$$

using which the properties of Data Bubbles can be computed when required.

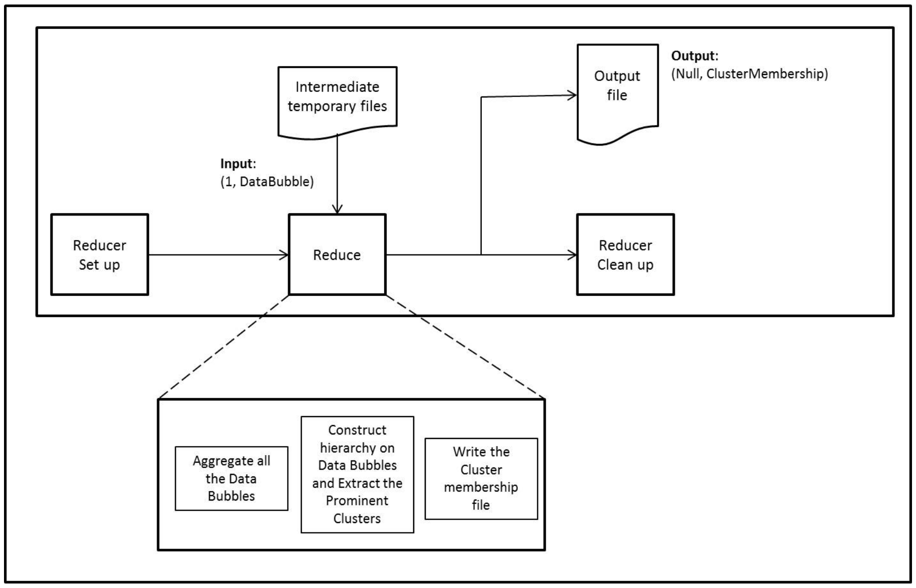
Intermediate temporary files
Output file
Output:
(Null, ClusterMembership)
Input:
(1, DataBubble)
Reducer Set up
Reduce
Reducer Clean up
Aggregate all the Data Bubbles
Construct hierarchy on Data Bubbles and Extract the Prominent Clusters
Write the Cluster membership file

Figure 7.14: Layer 1 Reducer in the Recursive Sampling Approach using Data Bubbles

After aggregating all the Data Bubbles into set B, HDBSCAN* hierarchy is computed on B w.r.t m_{pts} by computing a minimum spanning tree on the mutual reachability graph of B. Extract the clusters on this hierarchy. Find the *inter-cluster edges* from the computed minimum spanning tree. The output of the reducer is a *"Bubble Cluster Membership file"* F_B and a set of inter-cluster edges *"Inter-Cluster Edges"*, which are written to the file system. F_B contains all the Data Bubbles along with their cluster membership. Any Data Bubble which was considered noise is discarded and not added to F_B.

At Driver Node: The Driver Node, creates another job with the same *data splits* of Layer 1 but with the *"Cluster Membership file"* added to the *distributed cache*. The mapper and the reducer of Layer 2 are called *"PHDBSCAN Mapper"* (illustrated in figure 7.3.2) and *"PHDBSCAN Reducer"* (illustrated in figure 7.3.2) respectively.

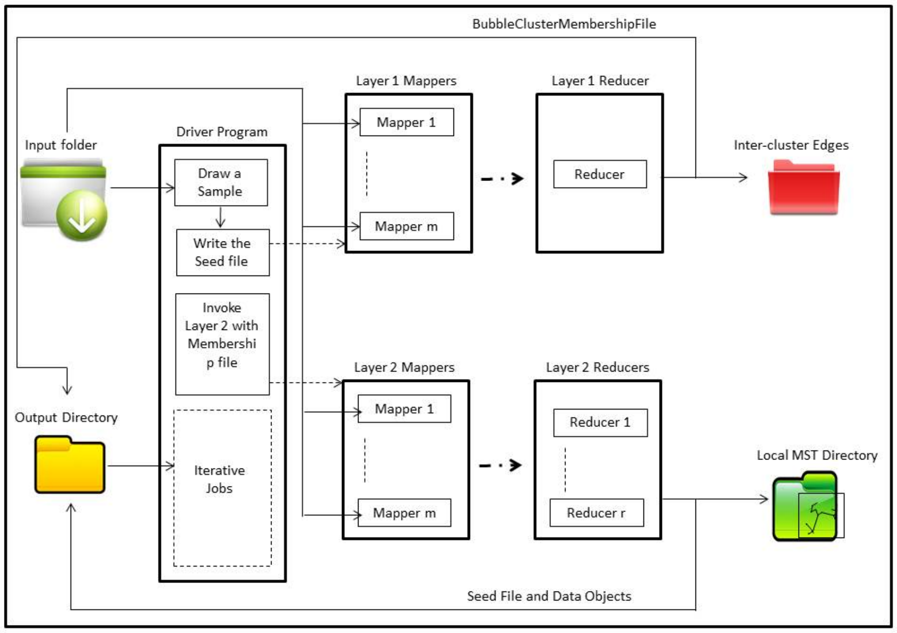

Figure 7.15: Master Processing Node in the Recursive Sampling Approach using Data Bubbles

Layer 2. PHDBSCAN Mapper: Each PHDBSCAN Mapper receives a data split and *"Bubble Cluster Membership file"*, F_B. Each map function receives the input in the form of $(key, value)$ pair with key as a unique identifier of a data split and $value$ is a data object. For each object x_p received by the map function, find the cluster membership of the Data Bubble to which the data object belongs and assign x_p to the same cluster. The PHDBSCAN Mapper at Layer 2 is shown in figure 7.3.2.

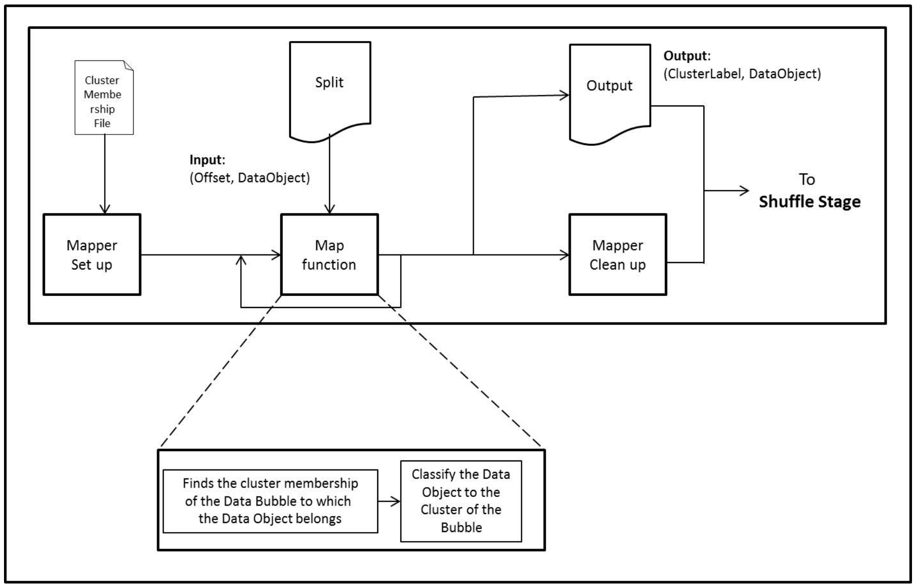
Cluster Membe rship File
Split
Input: (Offset, DataObject)
Output
Output: (ClusterLabel, DataObject)
To Shuffle Stage
Mapper Set up
Map function
Mapper Clean up
Finds the cluster membership of the Data Bubble to which the Data Object belongs
Classify the Data Object to the Cluster of the Bubble

Figure 7.16: Layer 2. PHDBSCAN Mapper in the Recursive Sampling Approach using Data Bubbles

Layer 2. PHDBSCAN Reducer: The Reducer is similar to the Recursive Sampling Reducer of PHDBSCAN implementation using only sampled objects explained in section 7.3.1 but with a modification. Instead of creating a hierarchy file when the size of the dataset received by the receiver exceeds the processing capacity τ, it just draws a sample which will be considered as *Seed set* for future iterations and writes to the file system as *"Seed file"*, F_{SS}. The flow of events within the PHDBSCAN Reducer at Layer 2 is shown in figure 7.3.2

Driver Node: Once the Reducer returns successfully, the driver checks for any output directory to be processed. If there are any directories to be processed, the driver creates another pair of jobs that run one after another, with each calling the mappers and reducers of Layer 1 and Layer 2 respectively. Once all the directories are processed, the edges are aggregated to form the combined MST, $MST_{combined}$ and HDBSCAN* hierarchy is computed. The algorithms are explained in Algorithm 19, 20, 21, 22 and 23, and the flow is explained in figure 7.12.

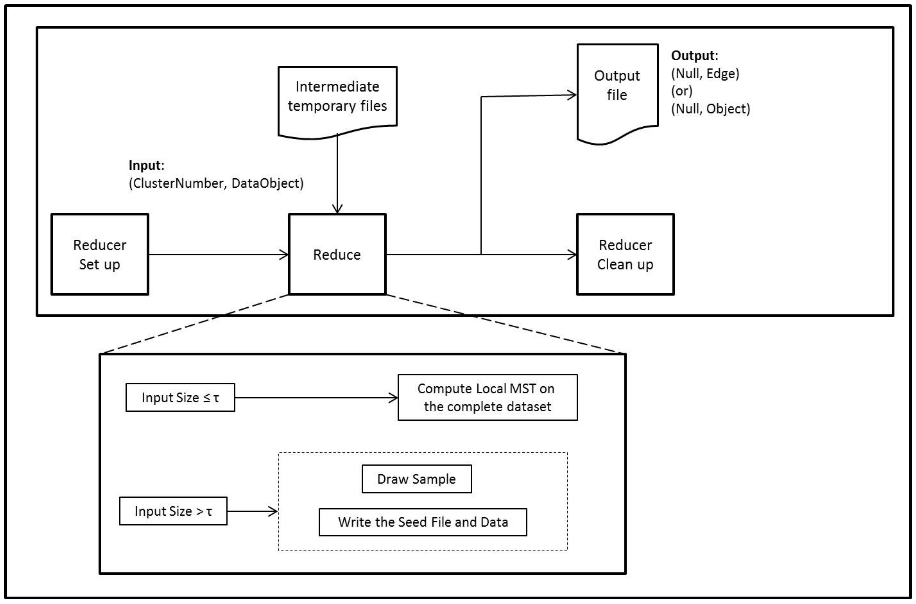

Figure 7.17: Layer 2. PHDBSCAN Reducer in the Recursive Sampling Approach using Data Bubbles

Chapter 8

Experiments and Results

In this chapter, we will present an experimental evaluation of the proposed parallelization approaches and compare them with HDBSCAN* in terms of the quality of results and the execution time. The experiments are conducted on various datasets in both pseudo-distributed and distributed environments.

8.1 Cluster Validation Measures

This section explains the clustering validation measures used in our experiments.

Adjusted Rand Index: The Adjusted Rand Index is based on the Rand Index or Rand Measure [40], which is a measure of similarity between two data partitions/clusterings. The Rand Index lies between 0 and 1. When the two partitions agree perfectly, the Rand Index is 1. The Rand Index can be defined in the following way:

Given a set of n objects $X = \{x_1, ..., x_n\}$ and two partitions of X, R and S. $R = \{r_1, ..., r_p\}$ and $S = \{s_1, ..., s_q\}$, the Rand Index is given by

$$Rand\ Index = \frac{a+d}{a+b+c+d}$$

where a,b,c and d are given by

- a = the number of pairs of elements in X that are in the same set in R and in the same set in S.

- c = the number of pairs of elements in X that are in the same set in R and in different sets in S.
- b = the number of pairs of elements in X that are in different sets in R and in the same set in S.
- d = the number of pairs of elements in X that are in different sets in R and in different sets in S.

The **Adjusted Rand Index (ARI)** is a version of Rand Index that is corrected for chance, which was proposed in [41], and which assumes the generalized hypergeometric distribution as the model of randomness. The ARI yields negative values if the index is less than the expected index. The general form for adjustment of an index by chance, for an index with a constant expected value, is given by

$$Adjusted\ Rand\ Index = \frac{Index - Expected\ Index}{Max\ Index - Expected\ Index}$$

Given the definition for the Rand Index, the *Adjusted Rand Index* is defined by

$$ARI = \frac{a - \frac{(a+c)(a+b)}{a+b+c+d}}{\frac{2a+b+c}{2} - \frac{(a+c)(a+b)}{a+b+c+d}}$$

where a, b, c and d are defined as above for the Rand Index.

Jaccard Index: The Jaccard Index or Jaccard Similarity coefficient is another measure of similarity between two partitions. The jaccard co-efficient is given by

$$J = \frac{a}{a+b+c}$$

where a, b and c are defined as above for the Rand Index.

F-Measure (F1 Score): The F-measure or the F1 Score is an evaluation measure based on the harmonic mean of precision and recall. Precision is defined as the fraction of objects that are correctly assigned to a cluster, and Recall is the extent to which a cluster contains all the objects of a specified class.

$$Precision, P = \frac{TP}{TP + FP}$$

$$Recall, R = \frac{TP}{TP + FN}$$

$$F\text{-}measure = \frac{2 \times P \times R}{P + R}$$

where TP = number of True Positives, FP = number of False Positives and FN = number of False Negatives.

Fowlkes-Mallows Index: The Fowlkes-Mallows [42] index is an external validation measure to determine the similarity between two partitions. Fowlkes Mallows is given by

$$FM = \sqrt{\frac{TP}{TP + FP} \times \frac{TP}{TP + FN}}$$

where TP, FP, FN are same as defined for F-Measure.

We report the following four cluster validation measures for all experiments to evaluate the quality of results: Adjusted Rand Index, Jaccard Co-efficient, F-Measure and Fowlkes-Mallows.

8.2 Experimental Setup

Pseudo-distributed system: Some of the experiments, designed to test the quality of PHDBSCAN clustering solutions, are run on a local machine and implemented using the Hadoop implementation of the MapReduce Framework (version 2.4.0). The distributed components of the MapReduce Framework are invoked sequentially, starting from the *driver*, followed by the *mappers* and *reducers* (all the mappers and reducers would run sequentially) before returning to the *driver* for final processing and termination.

Implementing the algorithm in MapReduce on a single machine satisfies the following conditions:

- The order of execution of various components in the framework is same as the MapReduce on a distributed environment. For example, no reducer is started before the successful completion of all the mappers.
- Memory is not shared between different components. Although machine and hardware for the execution of different tasks is the same, the objects are not shared between different tasks. This means that the data on which a mapper is executed is not available to another mapper (the same condition holds good for reducers too), thus simulating a distributed environment.

Distributed Environment: To identify the improvements on execution time of PHDBSCAN over HDBSCAN*, a true distributed architecture is used. For these experiments, the algorithms are run on the Hadoop implementation of the MapReduce framework on Amazon Elastic MapReduce (EMR). The version used for our experiments is Hadoop 2.4.0 (AMI version 3.3.1). We use Amazon Simple Storage Service (S3) as the object storage system to store the results and data. Experiments on Amazon EMR were conducted with three different types of instances (nodes) with the following configurations:

Instance Type	vCPU	Memory(GiB)	SSD Storage(GiB)
m1.medium	1	3.75	1×4
m1.large	2	7.5	1×32
c3.xlarge	4	7.5	2×40

Whenever Amazon EMR is used for conducting an experiment, the types of instances used for a set of experiments are mentioned explicitly in the section describing the experiment. In a distributed environment, two broadly classified node types are used; *"master node"* and *"core node"*. A *master node* is where the *driver* runs and *core node* is one of the distributed nodes that operate in parallel (for mappers and reducers).

8.3 Evaluation of the Quality of Results

Different combination of approaches can be used to extract the final clustering results of PHDBSCAN. Figure 8.1 illustrates the various combinations of the possible approaches where results can be extracted differently by following a unique path.

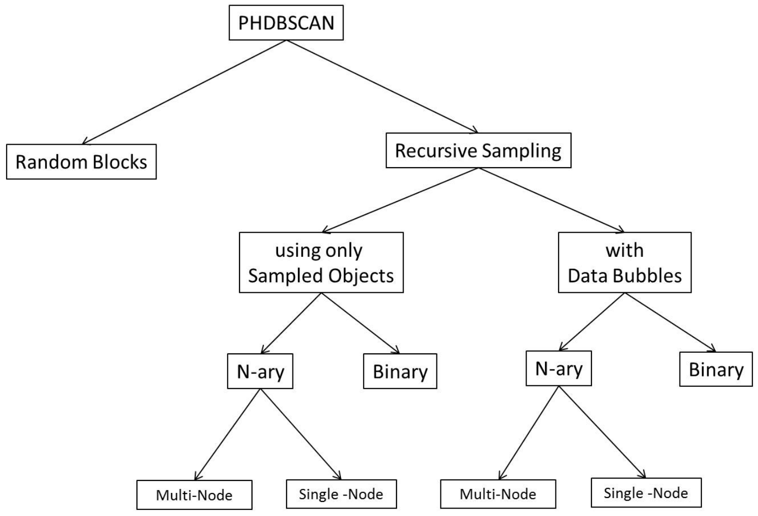

Figure 8.1: List of all possible approaches possible in PHDBSCAN

8.3.1 Binary vs N-ary Approach

This section shows the comparison of the *Binary method* with the *N-ary method* of cluster extraction in *PHDBSCAN Recursive Sampling Approach using Data Bubbles*. The *Binary approach* extracts the first formed prominent clusters (at least two clusters) while the *N-ary approach* traverses the complete hierarchy to find all the prominent clusters. Experiments were conducted on a dataset with 20000 2-dimensional data objects. The data was generated by a data generator with a mix of 19 randomly generated gaussian distributions and 1000 objects generated in random within the data space. The PHDBSCAN parameters were set to $m_{pts} = m_{clSize} = 5$ and $\tau = 1000$. Experiments are repeated with 15 independently selected sample sets. Clustering results of HDBSCAN* with same m_{pts} and m_{clSize} are used as the ground truth for measuring various indexes. The results for various indexes and different sample sizes are shown in figures 8.2 through 8.5. The results show that for both *Binary* and *N-ary* cluster extraction, the clustering results are very similar or almost identical. This is due to the fact that the clustering solution of the PHDBSCAN hierarchy is not dependent on the depth of the recursion

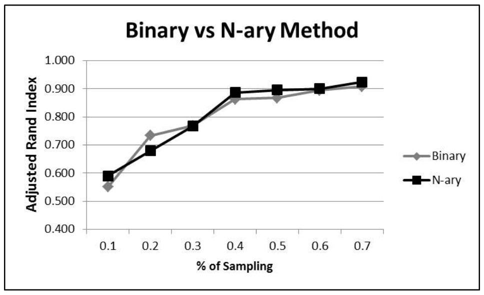

Figure 8.2: Adjusted Rand Index

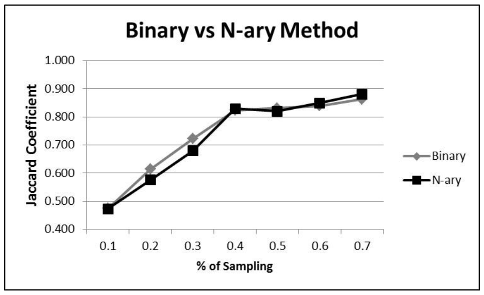

Figure 8.3: Jaccard Coefficient

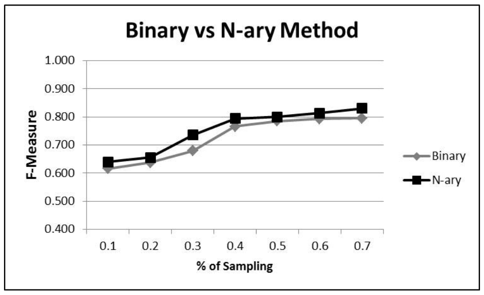

Figure 8.4: F-Measure

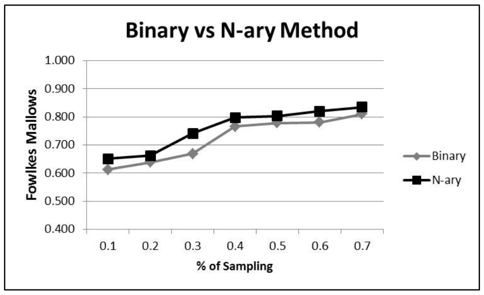

Figure 8.5: Fowlkes Mallows Score

at which a cluster (that is a part of the final clustering solution) is formed. The final clustering solution will still comprise of the same clusters extracted irrespective of the approach.

8.3.2 Multi-Node vs Single-Node Cluster Extraction

Multi-Node cluster extraction is a method of forming a cluster tree based on the clusters extracted at distributed nodes using Data Bubbles. Multi-Node Cluster extraction is explained in detail in section 6.4.1. Single-Node cluster extraction is extracting the most prominent clusters on the combined MST at the *Driver* node. Experiments were conducted on a dataset with 9000 10-dimensional data objects. The data was generated by the same data generator with a mix of 17 randomly generated gaussian distributions and 500 data objects generated in random within the data space. The PHDBSCAN parameters were set to $m_{pts} = m_{clSize} = 5$ and $\tau = 1000$. Experiments are repeated with 15 independently selected sample sets. Clustering results of HDBSCAN* with same m_{pts} and m_{clSize} are used as the ground truth. The quality of the results for different sample sizes are shown in figures 8.6 through

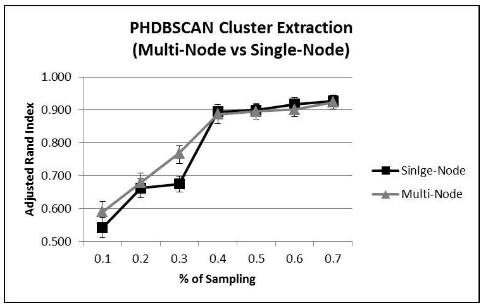

Figure 8.6: Adjusted Rand Index

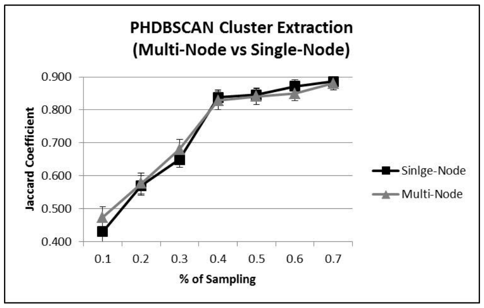

Figure 8.7: Jaccard Coefficient

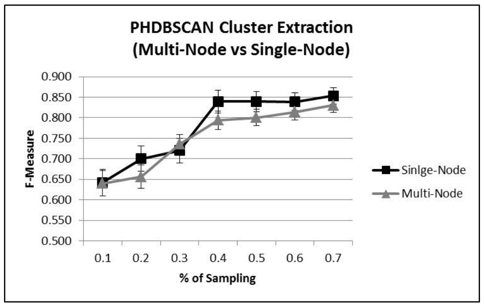

Figure 8.8: F-Measure

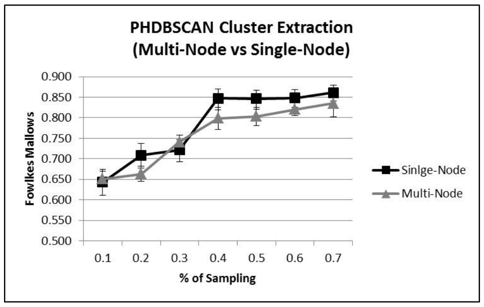

Figure 8.9: Fowlkes Mallows Score

8.9. The results show that for both *Multi-node* and *Single-node* extraction, the clustering solutions tend to converge after certain sampling rate since the stabilities captured in a *distributed cluster tree* (explained in section 6.4.1) reflect the stabilities of the clusters in the final cluster tree formed at a single node.

Different datasets are evaluated to compare the clustering solutions produced by HDBSCAN* and the PHDBSCAN algorithms. By default, and unless explicitly stated otherwise, all experiments to compare the quality of results are conducted using the PHDBSCAN Recursive Sampling Approach (both using sampled objects and Data Bubbles) with the N-ary method of finding prominent clusters at each recursive step and final prominent clusters are extracted at a single node. The experiments were repeated 15 times with 15 randomly and independently selected sample sets and the average values for all the validation measures are reported in the figures.

The first set of experiments compares the performance of the methods on

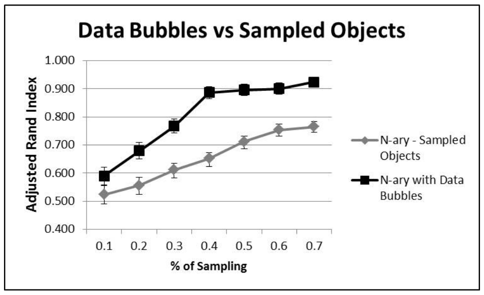

Figure 8.10: Adjusted Rand Index

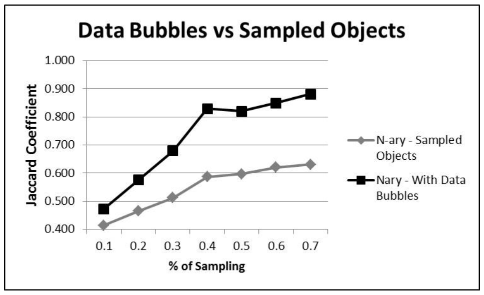

Figure 8.11: Jaccard Coefficient

datasets with well separated clusters. Two datasets were generated with clearly defined clusters. The first dataset was generated with 7 clusters of 20000 2-dimensional data objects with no *noise* objects. The data was generated with a mix of gaussian distributions such that there are 7 well-separated clusters of different shapes. Both algorithms, HDBSCAN* and PHDBSCAN, clearly identified all 7 clusters with no object identified as a *noise* object. The second dataset consisted of 100000 8-dimensional data objects with 7 clusters. For both experiments, parameters were set at $m_{pts} = 10$ and $m_{clSize} = 100$. Sample sets were drawn at 0.5% sampling rate. The results were identical for both HDBSCAN* and PHDBSCAN (both using only sampled objects and Data Bubbles) identifying 7 clusters without any noise objects.

8.3.3 Recursive Sampling Approach: Data Bubbles vs Only Sampled Objects

To compare the quality of results of PHDBSCAN using only sampled objects with that of PHDBSCAN using Data Bubbles (explained in detail in sections 5.2 and 6.4 respectively), we conducted experiments on a dataset with the 20000 2-dimensional data objects used in section 8.3.1. The parameters were set to $m_{pts} = m_{clSize} = 5$ and $\tau = 1000$. Different validation measures are shown in figures 8.10 through 8.13. It is inferred from the figures that using Data Bubbles on the dataset improves the accuracy of the clustering results.

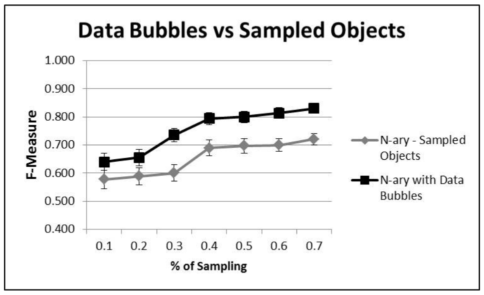

Figure 8.12: F-Measure

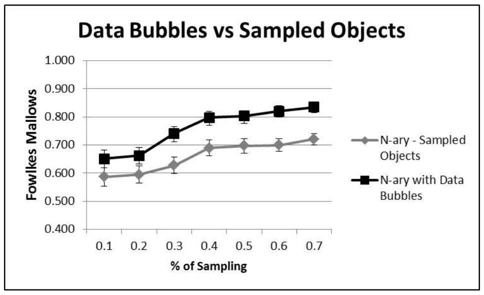

Figure 8.13: Fowlkes Mallows Score

8.3.4 Quality of Results on Real Datasets

To test the accuracy on algorithms with real datasets, three different datasets were used.

IRIS: *Iris* dataset is a part of the UCI repository [43] and it is one of the most used datasets in the pattern recognition literature. The dataset contains 3 classes of 50 instances each with 4 features, where each class contains a type of iris plants described by the length and the width of their petals and sepals. Experiments were conducted with parameters $m_{pts} = m_{clSize} = 4$ for both PHDBSCAN and HDBSCAN* (similar to the parameters used in [7]). Sample sets consisted of at least 21 objects. The results for different validation measures are shown in figure 8.14. The results show that the PHDBSCAN with both sampled objects and Data Bubbles give similar measures. This is due to the fact that the dataset is small and the size of the sample set (21 out of 150) is relatively large that both methods converge to the same measures. Smaller sample sets will have the tendency to be skewed unless the sample is carefully chosen to accommodate equal samples from all three possible classes.

Gas Sensor Array Drift Dataset: The *"Gas Sensor Array Drift Dataset"* [44] consists of 13910 observations from 16 chemical sensors utilized in simulations for drift compensation in a discrimination task of 6 gases at various levels of concentration. The dataset is gathered for a period of 36 months in a gas delivery platform facility situated at the ChemoSignals Laboratory in the BioCircuits Institute, University of California San Diego. Each observation consists of a 128 dimensional feature vector across 16 sensors, each described

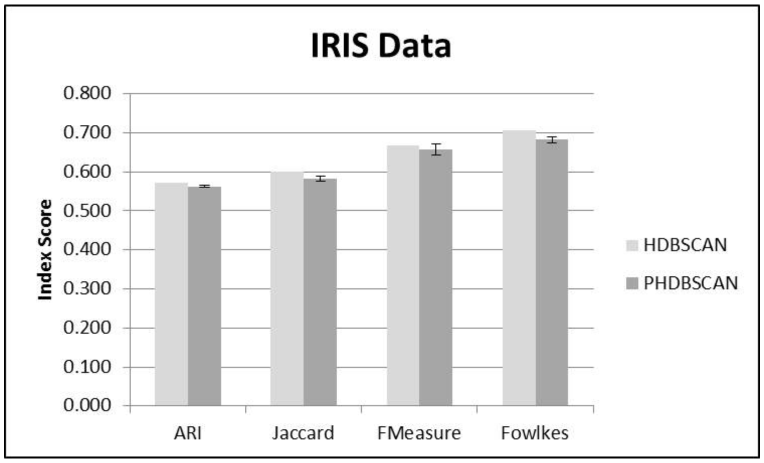

Figure 8.14: Accuracy of IRIS Data (Comparison of HDBSCAN* and PHDBSCAN-Recursive Sampling Approaches)

by 8 features. The dataset contains of 6 distinct classes of 6 different gaseous substances, namely Ammonia, Acetaldehyde, Acetone, Ethylene, Ethanol, and Toluene, dosed at a wide variety of concentration levels. The objective of a clustering method is to find these distinct classes by identifying the state of the levels of the gases across various sensors. Experiments were conducted with parameters $m_{pts} = m_{clSize} = 5$. The results for the different validation measures are shown in figure 8.15. The results of PHDBSCAN using Data Bubbles are better than the PHDBSCAN using only sampled objects.

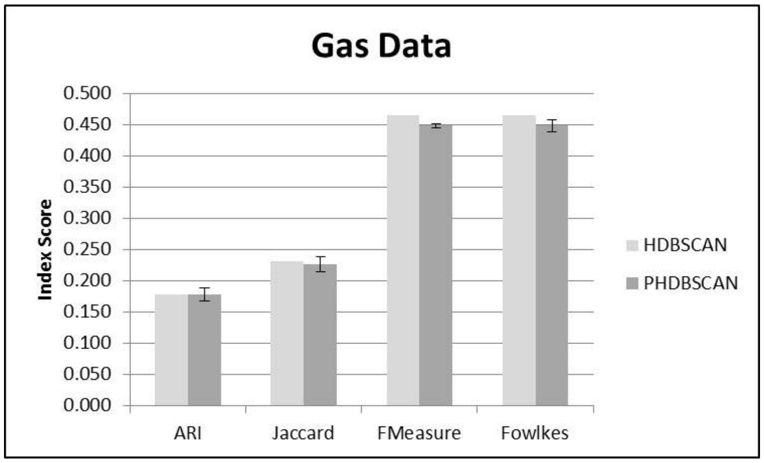

Figure 8.15: Accuracy of Gas Data (Comparison of HDBSCAN* and PHDBSCAN-Recursive Sampling Approaches)

YouTube Multiview Video Games Dataset: The third dataset is the *"YouTube Multiview Video Games Dataset"* [45]. This dataset consists of feature values and class labels for about 120000 videos (instances). Each instance is described by up to 13 feature types, from 3 high level feature families: *textual*, *visual* and *auditory* features. The dataset contains 31 class labels, corresponding to popular video games. Out of the available 13 feature families with thousands of features, 87 video features were selected for experiments which were available for 95% of the dataset. The missing values were replaced with the Expectation-Maximization algorithm using IBM SPSS Statistics Desktop version 22.0 tool. The results are shown in figure 8.16. The results show that for this dataset, the PHDBSCAN - Recursive Sampling Approach with Data Bubbles approximate the solutions of HDBSCAN* than the PHDBSCAN - Recursive Sampling Approach with only sampled objects.

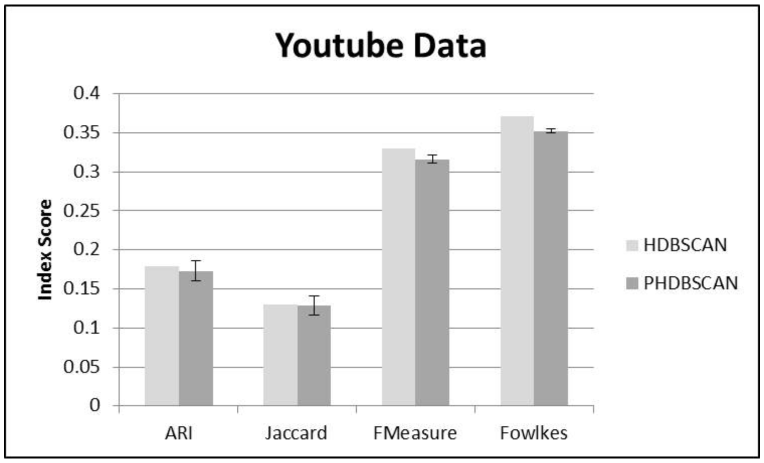

Figure 8.16: Accuracy of Youtube Data (Comparison of HDBSCAN* and PHDBSCAN-Recursive Sampling Approaches)

8.3.5 Influence of Node Capacity

The τ gives a measure of the maximum number of objects that can be processed by a node. The threshold can be set based on the hardware used. To measure the influence of the parameter τ, we conducted experiments on the youtube dataset with $m_{pts} = 10$ and $m_{clSize} = 100$. The results for values of τ between 1000 and 20000 are shown in figure 8.17. The results show that there is almost no difference in performance, indicating very little dependence on the parameter τ.

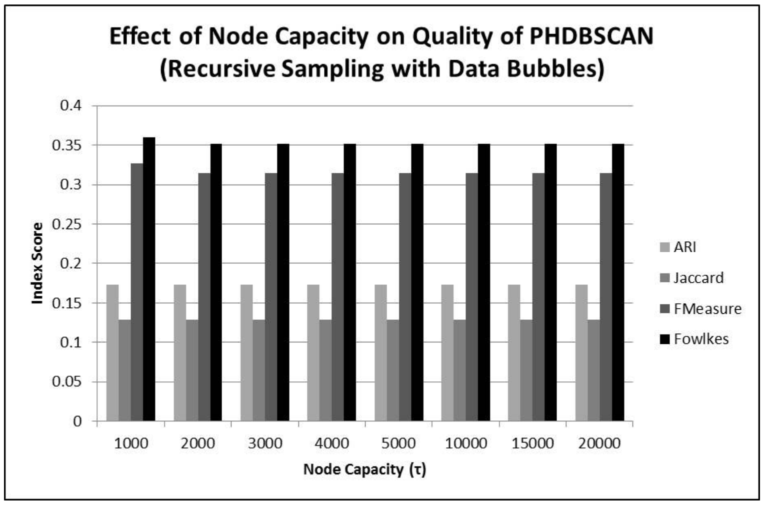

Figure 8.17: Influence of PerNodeThreshold on Accuracy

8.4 Evaluation of Execution Time

To evaluate the runtime of our proposed methods, the following experiments are run on Amazon EMR with S3 for storing objects. The MapReduce parameters for speculative execution are set to *off* at both task level and job level.

8.4.1 Random Blocks

The *Random Blocks Approach* is an approach that randomly divides the data into multiple overlapping datasets called *data blocks*. Each data block is processed independently and finally combined at a single reducer. The whole process is divided into two parts

- Independently finding the local Minimum Spanning Tree (MST) at every node.
- Combining the local MSTs into a single overall MST.

The experiments to evaluate the execution time of the Random Blocks Approach were run on Amazon Elastic Cloud Computing using the Amazon EMR framework. The dataset consisted of 1 million 2-dimensional data objects. The experiments are run on a distributed environment with 10 core nodes and 1 master node, all instances of type m1.medium. The 10 core nodes are responsible for calculating the local Minimum Spanning Trees and one of them is used as the reducer to combine all the local MSTs.

We measured the execution time of the dataset for different values of m_{pts} varying from 2 to 5 with $m_{clSize} = 1000$ and $\tau = 100000$. The figure 8.18 shows the increase in processing time with respect to the change in m_{pts}. The processing time for the *Random Blocks* approach works well for smaller values of m_{pts} and for $m_{pts} > 4$, the execution time exceeds the execution time of HDBSCAN* algorithm on a single node. This is due to the fact that the number of data blocks created increases considerably with a small change in m_{pts} and the total number of edges aggregated by all the local MSTs increase with increase in the number of local MSTs. This behaviour severely limits the applicability of the Random Blocks Approach to speed up density-based hierarchical clustering. Values for $m_{pts} > 4$ are rather the rule than the exception in applications of density-based clustering.

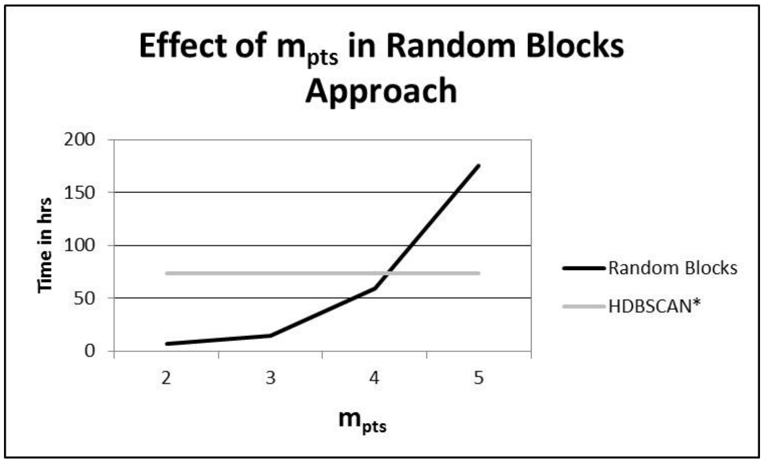

Figure 8.18: Comparison of Execution time of the Random Blocks Approach with HDBSCAN* on different values of m_{pts}

8.4.2 Recursive Sampling Approach with Data Bubbles using the N-ary extraction method

Set of experiments was conducted to compare the execution time of PHDBSCAN using the N-ary extraction method for datasets of different sizes. We used 6 synthetic datasets with 1, 2, 3, 4, 5 and 10 million data objects, respectively, and at 0.5% sampling rate. The experiments were conducted with 10 different samples using 3 c3.xlarge instances as core nodes and a c3.xlarge instance as master node. The average runtime over the 10 different samples is shown in figure 8.19. It is observed that increase in execution time with respect to the increase in size of the input is much lesser than the usual quadratic run time complexity of HDBSCAN*.

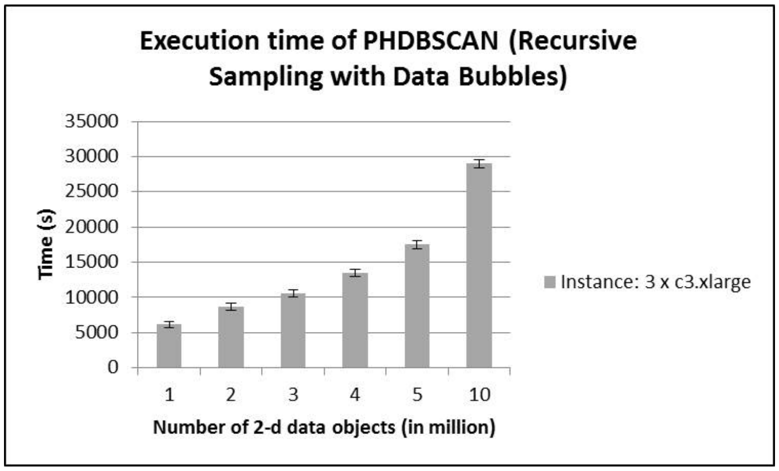

Figure 8.19: Comparison of Execution time for different datasets of increasing sizes using N-ary Approach

A set of experiments was also conducted to study the influence of the number of distributed core nodes, given the same dataset with same the sampling rate. Figures 8.20 and 8.21 show the execution time on 2 synthetic datasets, consisting of 1 million and 10 million data objects, respectively. From figure 8.20 and 8.21, we can infer that the performance gain is initially large by adding more core nodes but with addition of more core nodes, the performance gain is marginal. This is due to the fact that the number of required reducers (where the sequential aspect of the algorithm runs and consumes the maximum time) are dependent on the number of clusters that are extracted at each level.

Therefore, an increase in number of core nodes may render some of the core nodes to be idle at some iterations, depending on the number of extracted clusters. The marginal increase in performance is due to the fact that the mappers, that are operated in parallel, are dependent on the number of data splits (each data split goes to a mapper as discussed in section 7.3.1) and in the experiments that were conducted, the number of data splits is always larger than the number of distributed nodes.

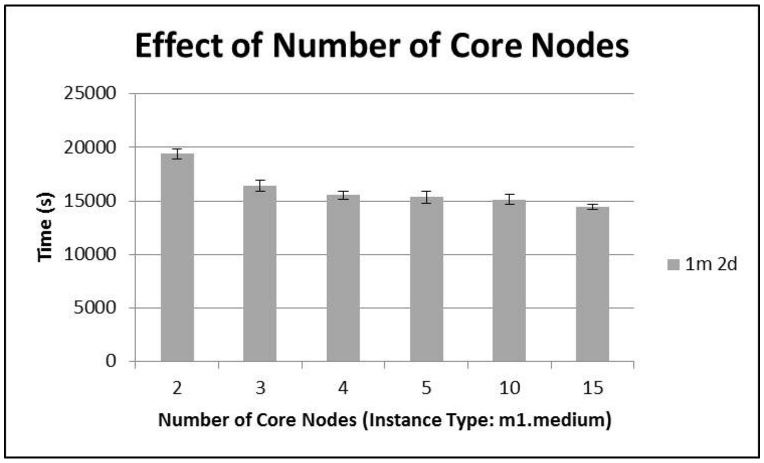

Figure 8.20: Influence of number of distributed nodes on execution time of 1 million data objects using m1.medium instances of worker nodes

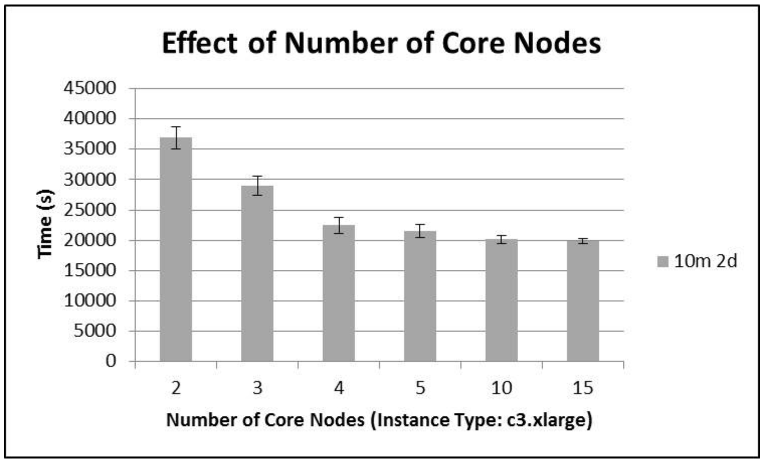

Figure 8.21: Influence of number of distributed nodes on execution time of 10 million data objects using c3.xlarge instances of worker nodes

8.4.3 Comparing with the Binary method of cluster extraction

We also conducted a set of experiments to evaluate the execution time of the *PHDBSCAN Recursive Sampling Approach* with Data Bubbles using the Binary method of extraction. Experiments were conducted with 10 independent sample sets at 0.5% sampling rate for three different datasets using both *N-ary* and *Binary* methods. The first two sets of experiments were conducted with two different datasets consisting of 1 million data objects, described by 2-dimensional and 10-dimensional feature vectors, respectively. These first two sets of experiments were conducted with 4 core nodes on the m1.medium instance. The third set of experiments was conducted on the same 1 million 2-dimensional data, but on the c3.xlarge instance. Figure 8.22 illustrates that

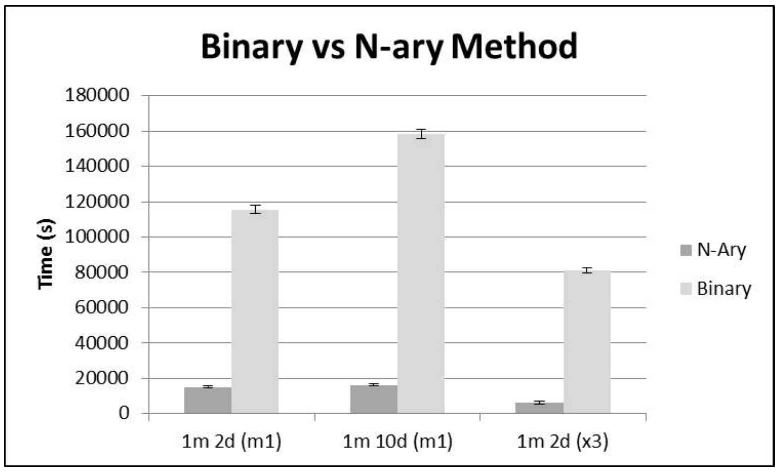

Figure 8.22: Comparison of Binary Approach and N-ary Approach for different datasets

the execution time using the *Binary* method is many times larger than the execution time using the *N-ary* method of cluster extraction using the same parameters and core node configuration. This is due to the fact that in Binary method, at all levels of iterations, the number of clusters extracted are almost always 2. This increases the number of iterations that are required to complete the execution of the *Recursive Sampling Approach* of PHDBSCAN using the *Binary* method. Adding more iterations to the processing induces additional job setup time, time to assign nodes for particular files, time for the shuffle stage for more rounds and data I/O.

8.5 Conclusions

MapReduce framework is a programming paradigm that processes massive amounts of unstructured data in parallel across a distributed cluster. Hierarchical clustering algorithms, that have many advantages over the traditional partitional clustering algorithms, were difficult to operate in parallel due to their constraints. In this dissertation we introduced two different approaches to parallelize the hierarchical density-based clustering algorithm called HDBSCAN* by data parallelism. The first method called the *"Random Blocks Approach"* finds an exact HDBSCAN* hierarchy while the second method called *"Recursive Sampling Approach"* finds an approximate version of HDBSCAN* hierarchy. These approaches were implemented using the MapReduce framework by converting the recursive problem of dividing the data to a MapReduce problem, allowing the hierarchical clustering algorithm to run on parallel machines.

We also evaluated different validation measures in terms of the quality of results, and also measured the improvement in execution time of the algorithms. The quality of results show that PHDBSCAN Recursive Sampling Approach, approximately identifies the clustering partitions generated by the HDBSCAN* algorithm. Implementing the Recursive Sampling approach with a data summarization technique called Data Bubbles, improves the accuracy of the clustering solutions.

We first evaluated the execution time to construct the exact HDBSCAN* hierarchy using the Random Blocks Approach with varying m_{pts}. The Random Blocks Approach fails to improve the execution time using parallel systems due to the increase in the number of files to be processed and repeatedly computing edges between same vertices at multiple nodes. Therefore, the Random Blocks Approach does not really take advantage of the available parallel processing units.

The parallel systems are best used using the Recursive Sampling Approach. We showed empirically, the improvements in execution time of PHDBSCAN using Recursive Sampling Approach on parallel processing units over the HDBSCAN* algorithm on a single processing unit. The PHDBSCAN Recursive Sampling Approach is also scalable and the results were shown for dataset of upto 10 million data objects. The PHDBSCAN has a loss of quality in the results due to approximation; loss of quality is a trade-off for the time required to compute an exact HDBSCAN* hierarchy. Given the amount of time taken by the HDBSCAN* algorithm to run on large datasets, a quick clustering can be obtained using PHDBSCAN Recursive Sampling approach. We also proposed a method to quickly find a *Distributed Cluster Tree* which can be used to find the set of most prominent clusters without the need to combine the individual solutions from various distributed nodes and creating a hierarchy on the combined solution. This helps to quickly analyze the approximate clusters, and take decisions based on it.

The idea of using *Distributed Cluster Tree* can be extended to identify the outlier scores immediately. The original HDBSCAN* paper [7] introduces an outlier score for noise objects called *"Global-Local Outlier Score from Hierarchies"(GLOSH)*. These scores can be approximately computed using the Distributed Cluster Tree and thus approximately assigning a degree of outlierness for the noise objects extracted at various levels of density.

Bibliography

[1] C. Jin, M. M. A. Patwary, A. Agrawal, W. Hendrix, W.-k. Liao, and A. Choudhary, "Disc: A distributed single-linkage hierarchical clustering algorithm using mapreduce," *work*, vol. 23, p. 27.

[2] B. S. Everitt, S. Landau, and M. Leese, *Cluster Analysis*. Wiley Publishing, 4th ed., 2009.

[3] P. Hansen and B. Jaumard, "Cluster analysis and mathematical programming," *Math. Program.*, vol. 79, pp. 191–215, Oct. 1997.

[4] A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[5] L. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, no. 3, pp. 338 – 353, 1965.

[6] S. Johnson, "Hierarchical clustering schemes," *Psychometrika*, vol. 32, no. 3, pp. 241–254, 1967.

[7] R. J. Campello, D. Moulavi, and J. Sander, "Density-based clustering based on hierarchical density estimates," in *Advances in Knowledge Discovery and Data Mining*, pp. 160–172, Springer Berlin Heidelberg, 2013.

[8] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: An efficient data clustering method for very large databases," 1996.

[9] S. Guha, R. Rastogi, and K. Shim, "Cure: An efficient clustering algorithm for large databases," in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, (New York, NY, USA), pp. 73–84, ACM, 1998.

[10] S. G. Rajeev, R. Rastogi, and K. Shim, "Rock: A robust clustering algorithm for categorical attributes," in *Information Systems*, pp. 512–521, 1999.

[11] G. Karypis, E.-H. S. Han, and V. Kumar, "Chameleon: Hierarchical clustering using dynamic modeling," *Computer*, vol. 32, pp. 68–75, Aug. 1999.

[12] L. Kaufman and P. J. Rousseeuw, *Monothetic Analysis (Program MONA)*, pp. 280–311. John Wiley & Sons, Inc., 2008.

[13] R. Sibson, "SLINK: An Optimally Efficient Algorithm for the Single-Link Cluster Method," *The Computer Journal*, vol. 16, pp. 30–34, 1973.

[14] C. F. Olson, "Parallel algorithms for hierarchical clustering," *Parallel Computing*, vol. 21, pp. 1313–1325, 1995.

[15] H.-R. Tsai, S.-J. Horng, S.-S. Lee, S.-S. Tsai, and T.-W. Kao, "Parallel hierarchical clustering algorithms on processor arrays with a reconfigurable bus system.," *Pattern Recognition*, vol. 30, no. 5, pp. 801–815, 1997.

[16] C.-H. Wu, S.-J. Horng, and H.-R. Tsai, "Efficient parallel algorithms for hierarchical clustering on arrays with reconfigurable optical buses.," *J. Parallel Distrib. Comput.*, vol. 60, no. 9, pp. 1137–1153, 2000.

[17] S. Rajasekaran, "Efficient parallel hierarchical clustering algorithms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 6, pp. 497–502, 2005.

[18] E. M. Rasmussen and P. Willett, "Efficiency of hierarchic agglomerative clustering using the icl distributed array processor," *J. Doc.*, vol. 45, pp. 1–24, Mar. 1989.

[19] J. L. Bentley, B. W. Weide, and A. C. Yao, "Optimal expected-time algorithms for closest point problems," *ACM Trans. Math. Softw.*, vol. 6, pp. 563–580, Dec. 1980.

[20] J. L. Bentley and H.-T. Kung, "Two papers on a tree-structured parallel computer," Tech. Rep. CMU-CS-79-142, Carnegie-Mellon University.Computer science. Pittsburgh (PA US), 1979.

[21] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, "Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation," *Commun. ACM*, vol. 31, pp. 1343–1354, Nov. 1988.

[22] D.-J. Chang, M. M. Kantardzic, and M. Ouyang, "Hierarchical clustering with cuda/gpu.," in *ISCA PDCCS* (J. H. Graham and A. Skjellum, eds.), pp. 7–12, ISCA, 2009.

[23] Z. Du and F. Lin, "A novel parallelization approach for hierarchical clustering," *Parallel Computing*, vol. 31, no. 5, pp. 523 – 527, 2005.

[24] S. Wang and H. Dutta, "PARABLE: A PArallel RAndom-partition Based HierarchicaL ClustEring Algorithm for the MapReduce Framework," *The Computer Journal*, vol. 16, pp. 30–34, 2011.

[25] V. Olman, F. Mao, H. Wu, and Y. Xu, "Parallel clustering algorithm for large data sets with applications in bioinformatics," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 6, no. 2, pp. 344–352, 2009.

[26] W. Hendrix, M. Ali Patwary, A. Agrawal, W. keng Liao, and A. Choudhary, "Parallel hierarchical clustering on shared memory platforms," in *High Performance Computing (HiPC), 2012 19th International Conference on*, pp. 1–9, Dec 2012.

[27] W. Hendrix, D. Palsetia, M. Ali Patwary, A. Agrawal, W. keng Liao, and A. Choudhary, "A scalable algorithm for single-linkage hierarchical clustering on distributed-memory architectures," in *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*, pp. 7–13, Oct 2013.

[28] M. Ankerst, M. M. Breunig, H. peter Kriegel, and J. Sander, "Optics: Ordering points to identify the clustering structure," pp. 49–60, ACM Press, 1999.

[29] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of Computational and Applied Mathematics*, vol. 20, no. 0, pp. 53 – 65, 1987.

[30] M. Ester, H. peter Kriegel, J. S, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," pp. 226–231, AAAI Press, 1996.

[31] T. Pei, A. Jasra, D. Hand, A.-X. Zhu, and C. Zhou, "Decode: a new method for discovering clusters of different densities in spatial data," *Data Mining and Knowledge Discovery*, vol. 18, no. 3, pp. 337–369, 2009.

[32] G. Gupta, A. Liu, and J. Ghosh, "Automated hierarchical density shaving: A robust automated clustering and visualization framework for large biological data sets," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 7, pp. 223–237, Apr. 2010.

[33] J. A. Hartigan, *Clustering Algorithms.* New York, NY, USA: John Wiley & Sons, Inc., 99th ed., 1975.

[34] P. Bradley, U. Fayyad, and C. Reina, "Scaling clustering algorithms to large databases," pp. 9–15, AAAI Press, 1998.

[35] W. DuMouchel, C. Volinsky, T. Johnson, C. Cortes, and D. Pregibon, "Squashing flat files flatter," in *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, (New York, NY, USA), pp. 6–15, ACM, 1999.

[36] M. M. Breunig, H. P. Kriegel, P. Kröger, and J. Sander, "Data bubbles: quality preserving performance boosting for hierarchical clustering," in *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international con-*

ference on Management of data, (New York, NY, USA), pp. 79–90, ACM, 2001.

[37] M. M. Breunig, H.-P. Kriegel, and J. Sander, “Fast hierarchical clustering based on compressed data and optics,” in *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, PKDD ’00, (London, UK, UK), pp. 232–242, Springer-Verlag, 2000.

[38] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[39] H. Karloff, S. Suri, and S. Vassilvitskii, “A model of computation for mapreduce,” in *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’10, (Philadelphia, PA, USA), pp. 938–948, Society for Industrial and Applied Mathematics, 2010.

[40] W. Rand, “Objective criteria for the evaluation of clustering methods,” *Journal of the American Statistical Association*, vol. 66, no. 336, pp. 846–850, 1971.

[41] L. Hubert and P. Arabie, “Comparing partitions,” *Journal of classification*, vol. 2, no. 1, pp. 193–218, 1985.

[42] E. B. Fowlkes and C. L. Mallows, “A Method for Comparing Two Hierarchical Clusterings,” *Journal of the American Statistical Association*, vol. 78, no. 383, pp. 553–569, 1983.

[43] K. Bache and M. Lichman, “UCI machine learning repository,” 2013.

[44] A. Vergara, S. Vembu, T. Ayhan, M. A. Ryan, M. L. Homer, and R. Huerta, “Chemical gas sensor drift compensation using classifier ensembles, sensors and actuators b,” 2012.

[45] O. Madani, M. Georg, and D. A. Ross, “On using nearly independent feature families for high precision and confidence,” *Machine Learning*, vol. 92, pp. 457–477, 2013.

Appendices

Appendix A

MapReduce Algorithms

Algorithm 13: Random Blocks Approach: Driver

```
Input: Dataset $X$, Parameter $m_{pts}$, Number of partitions $k$
Output: HDBSCAN* Hierarchy
System: Master Node
  1. Divide the dataset $X$ into $k$ partitions.
  2. Generate $l = \binom{k}{2 \times m_{pts}}$ data blocks, each with a unique identifier.
  3. Call Algorithm 14 with a set of data blocks as input. The output is a
     set of Minimum Spanning Trees, $MST_{local} = \{MST_1, ..., MST_l\}$ as set
     of edges.
  4. Call Algorithm 15 with the set of local Minimum Spanning Trees
     $MST_{local}$. Output is a combined MST, $MST_{combined}$, as a set of edges.
  5. Extract HDBSCAN* hierarchy as a dendrogram from $MST_{combined}$.
```

Algorithm 14: Random Blocks Approach : Mapper

```
Input: Data Block containing dataset $Y$ and Unique Identifier $i$,
       Parameter $m_{pts}$
Output: Local Minimum Spanning Tree, $MST_i$ on file system
System: Set of Distributed Nodes
  1. For each object in $Y$, compute core distance w.r.t. $m_{pts}$
  2. Compute a Mutual Reachability Graph $G_{m_{reach}}$ using the core
     distances.
  3. Compute an MST, $MST_i$ on $G_{m_{reach}}$.
  4. For each edge in $MST_i$, the (key, value) pair is given by (1, e) where $e$
     is given by $(u, v, d_{m_{reach}})$.
```

Algorithm 15: Random Blocks Approach : Reducer

```
Input: Set of local Minimum Spanning Trees
Output: Combined MST
System: Single Reducer
    1. Add all edges from all the input MSTs to a set $E$ and sort $E$ according
       to weight of the edges in $E$.
    2. Initialize the overall combined MST, $MST_{combined}$ with no edges.
    3. Starting from the edge with the least weight, do the following:
        (a) Remove the edge from $E$ and add the edge to $MST_{combined}$.
        (b) If edge $e$ is a duplicate edge of $e' \in MST_{combined}$, remove the edge
            with the maximum weight between $e$ and $e'$.
        (c) If addition of edge $e$ induces a cycle in $MST_{combined}$, remove $e$
            from $MST_{combined}$.
```

```
Algorithm 16: Recursive Sampling Approach Driver
Input: Dataset $X$, Parameter $m_{pts}$, $m_{clSize}$, Processing Capacity $\tau$,
       Arguments input and output directories
Output: HDBSCAN* Hierarchy
System: Master Node
  1. Draw a random sample $S$ from $X$ such that $| S | \leq \tau$.
  2. Find the most prominent clusters on HDBSCAN* hierarchy on dataset
     $X$ w.r.t. $m_{pts}$ and $m_{clSize}$.
  3. Write the data objects in $S$ with their cluster membership in a
     hierarchy file $F_H$.
  4. Create a job with configuration files and add $F_H$ to distributed cache of
     MapReduce.
  5. Run the job (Executing Algorithm 17 and 18 one after another).
     Output is either a list of directories to be processed with a Hierarchy
     file and Inter-Cluster Edges directory (or) local MST directory.
  6. while there exists a directory in "Output Directory" to be processed do
       (a) Select an input directory $D$ from the list of unprocessed
           directories.
       (b) Create job with $D$ as input directory.
       (c) Add the corresponding Hierarchy file to distributed cache.
       (d) Run the job with the configuration files and arguments.
       (e) if returned with a "fail" status then
             Re-Sample the dataset and run the job again
           else
             Mark the directory $D$ as "processed".
           end
     end
  7. Add all the edges from the MSTs in the directory "local MST
     directory" and all the inter-cluster edges in the directory "Inter Cluster
     Edges" to a set of edges, $MST_{combined}$.
  8. Extract HDBSCAN* hierarchy as a dendrogram from $MST_{combined}$.
```

Algorithm 17: Recursive Sampling Approach Mapper

```
Input: Data Block with dataset $Y$, Hierarchy file $F_H$, Configuration file
Output: Set of $(key, value)$ pairs
System: Set of Distributed Nodes (Mappers)
  1. For each object $x_p$ in $Y$, find the nearest neighbor of $x_p$, 2-$NN(x_p)$,
     among the data objects in $F_H$.
  2. Identify the cluster membership of 2-$NN(x_p)$ and assign $x_p$ to the
     same cluster.
  3. Write the output in the form of $(key, value)$ pair as
     $(ClusterLabel, DataObject)$.
```

Algorithm 18: Recursive Sampling Approach Reducer

```
Input: Set of $(key, value)$ pairs, Configuration file
Output: Directory "local MST directory" (or)
Output: Output Directory, set of inter-cluster edges "Inter-Cluster
        Edges", "Hierarchy File"
System: Set of distributed nodes (Reducers)
  1. Aggregate all the $(key, value)$ pairs into set $Y'$.
  2. if $\mid Y' \mid \leq \tau$ then
       (a) Compute the Minimum Spanning Tree, $MST_{local}$, on the mutual
           reachability graph created on dataset $Y'$ w.r.t. $m_{pts}$.
       (b) Write the edges of $MST_{local}$ to one or more files in "local MST
           directory".
     else
       (a) Draw random sample $S$ from $Y'$.
       (b) Extract most prominent clusters on $S$ using HDBSCAN*
           hierarchy and write the cluster membership of objects in $S$ to
           "Hierarchy file".
       (c) Identify the inter-cluster edges and write the edges to "Inter
           Cluster Edges". Check for the optional sampling tolerance
           parameter $\varsigma$. If not satisfied, return "fail" as status.
       (d) Add the data objects to a unique output directory in a parent
           directory "Output Directory".
     end
```

Algorithm 19: Recursive Sampling Approach with Data Bubbles - Driver

```
Input: Dataset $X$, Parameter $m_{pts}$, $m_{clSize}$, Processing Capacity $\tau$,
       Arguments input and output directories
Output: HDBSCAN* Hierarchy
System: Master Node
  1. Draw a random sample $S$, called the "Seed set" from $X$ such that
     $| S | \leq \tau$.
  2. Write the data objects in $S$ to a cluster file $F_{SS}$.
  3. Create a job with configuration files and add $F_{SS}$ to distributed cache
     of MapReduce.
  4. Run the job (Executing Algorithm 20 and 21 one after another).
     Output is a file $F_B$ with aggregated Data Bubbles and their respective
     cluster membership.
  5. Create a job with configuration files and add $F_B$ to distributed cache of
     MapReduce.
  6. Run the job (Executing Algorithm 22 and 23 one after another).
     Output is either a list of directories to be processed with Seed file and
     Inter-Cluster Edges directory (or) a directory local MST directory.
  7. while there exists a directory in "Output Directory" to be processed do
       (a) Select an input directory $D$ among the list of unprocessed
           directories.
       (b) Create a job, Layer1Job with $D$ as the input directory and add
           the corresponding $F_{SS}$ to distributed cache.
       (c) Run the Layer1Job with the configuration file and input
           arguments. Output is the file $F_B$.
       (d) Create another job Layer2Job, with $D$ as the input directory and
           add $F_B$ to distributed cache and run it. Output is either a local
           MST or an unprocessed output directory with a Seed file, $F'_{SS}$.
       (e) Mark the directory $D$ as "processed".
     end
  8. Add all the edges from the MSTs in the directory "local MST
     directory" and all the inter-cluster edges in the directory "Inter-Cluster
     Edges" to a set of edges, $MST_{combined}$.
  9. Extract HDBSCAN* hierarchy as a dendrogram from $MST_{combined}$.
```

```
Algorithm 20: Layer 1 - Bubble Mapper
  Input: Data Block with dataset $Y$, Seed file $F_S$, Configuration file
  Output: Set of $(key, value)$ pairs
  System: Set of Distributed Nodes (Mappers)
    1. Initialize a Data Bubble for each seed object in $F_S$.
    2. For each object $x_p$ in $Y$, find the nearest neighbor of $x_p$, 2-$NN(x_p)$,
       among the seed objects in $F_S$.
    3. Update the corresponding Data Bubble.
    4. Write the output in the form of $(key, value)$ pair as $(1, DataBubble)$.
```

```
Algorithm 21: Layer 1 - Bubble Reducer
  Input: Set of $(key, value)$ pairs, Configuration file
  Output: "Cluster Membership file" $F_B$, set of inter-cluster edges
          "Inter-Cluster Edges"
  System: Single Reducer node
    1. Aggregate all the Data Bubbles with same Data Bubble identifiers to
       get a set of Data Bubbles $B$.
    2. Compute HDBSCAN* hierarchy on $B$ w.r.t. $m_{pts}$ and extract the
       prominent clusters.
    3. Find the inter-cluster edges and add them to "Inter-Cluster Edges".
    4. Write the Data Bubbles with their cluster membership to $F_B$.
```

```
Algorithm 22: Layer 2 - PHDBSCAN Mapper
  Input: Data Block with dataset $Y$, File $F_B$ containing set $B$ of Data
         Bubbles and their cluster membership, Configuration file
  Output: Set of $(key, value)$ pairs
  System: Set of Distributed Nodes (Mappers)
    1. For each object $x_p \in Y$ which belong to a Data Bubble $B_{x_p}$, find the
       cluster membership of $B_{x_p}$ in $B$ and assign $x_p$ to the same cluster.
    2. Write the output in the form of $(key, value)$ pair as
       $(ClusterLabel, DataObject\ x_p)$.
```

Algorithm 23: Layer 2 - PHDBSCAN Reducer

```
Input: Set of $(key, value)$ pairs, Configuration file
Output: Directory "local MST directory" (or)
Output: Output Directory, "Seed File"
System: Set of distributed nodes (Reducers)
  1. Aggregate all the $(key, value)$ pairs into set $Y$.
  2. if $\mid Y \mid \leq \tau$ then
      (a) Compute the Minimum Spanning Tree, $MST_{local}$, on the mutual
          reachability graph created on dataset $Y$ w.r.t. $m_{pts}$.
      (b) Write the edges of $MST_{local}$ to one or more files in the directory,
          "local MST directory".
     else
      (a) Draw random sample $S$ from $Y$.
      (b) Add the objects in $S$ to a Seed file, $F_S$.
      (c) Add the data objects to a unique output directory in the parent
          directory "Output Directory".
     end
```