Fast, Parallel Stream Clustering using Hadoop Online

Georgios Christopoulos

July 25, 2011

Contents

List of Figures

List of Tables

Acknowledgements

This thesis would not have been possible without the guidance and the help of several individuals who in one way or another contributed and offered their valuable assistance in the preparation and completion of this study.

First and foremost, my utmost gratitude to my advisor, Professor Minos Garofalakis who gave me the possibility to complete this thesis with his supervision, advise and guidance from the very early stage of this research.

I am grateful for his encouragement and precious contribution throughout the elaboration of this study. I would also like to thank him for giving me the opportunity to work on this very interesting field of research. I am indebted to him more than he probably knows.

I would like to thank Assistant Professors Antonios Deligiannakis and Michael Lagoudakis who agreed to evaluate my diploma thesis. Moreover, I would like to thank my laboratory colleagues for their patience and constructive comments.

Also, I would like to thank all my friends for these great years we spent together and for many wonderful memories.

Most of all, I would like to thank my family for their enormous help, understanding and support throughout all these years as a student.

Abstract

In real-world problems we facing multi-dimensional and complex data that creates the need of classifying them into groups according to some common characteristics they have. This process of dividing data into groups is called clustering. Clustering is the assignment of a set of observations into subsets (called clusters) so that observations in the same cluster are similar in some sense. Clustering is a method of unsupervised learning, and a common technique for statistical data analysis used in many fields, including machine learning, data mining, pattern recognition, image analysis, information retrieval, and bioinformatics. Also, a serious problem that someone has to handle when works on real-world datasets is that they are massive and evolve over time. So, a motivation for our thesis was to work with efficiency on streams. In this thesis, we present the novel design and implementation of two stream clustering algorithms in the highly-scalable Map/Reduce parallel framework, using the Hadoop Online Prototype open-source implementation. Our experimental results with several large, real-life datasets on SoftNets HOP cluster verify the effectiveness of our approach.

Chapter 1

Introduction

Cluster Analysis suggests how groups of units are determined such that units within groups are similar in some respect and unlike those from other groups. Units in computer science would be any kind of multi-dimensional points. So, clustering is very useful in a large variety of applications in real world, such as biology, medicine, neuroscience, education, market etc. ,as points can be modeled by different forms of associations among entities. Examples in medical imaging, such as PET scans, cluster analysis can be used to differentiate between different types of tissue and blood in a three dimensional image; in market research when working with multivariate data from surveys and test panels and maybe partition the general population of consumers into market segments and to better understand the relationships between different groups of consumers/potential customers; in educational research analysis, data for clustering can be students, parents, sex or test scores.

One problem with the study of real world datasets is that they are massive and evolve over time. Obviously, it is difficult to apply sequential algorithms to analyze these data. This size constraint has led to the development of parallelization architectures. One recent and effective framework that permits the development of parallelized algorithms is Hadoop. Hadoop provides us with a distributed filesystem and the implementation of the map/reduce programming model, as well as all the necessary libraries that are needed in order for a compute cluster to function. Its main advantage is that it separates the parallelization code from the business logic, thus making easy for anyone to create and execute a parallel algorithm. Additionally it poses no restrictions regarding the number of computer nodes that the cluster should have, something that has been an issue in older architectures. However, Hadoop is not able to handle stream data, as input. Recently proposed extensions of Hadoop (e.g., Hadoop Online Prototype [6]) allow data to be pipelined between jobs and handle the continuous incoming data.

In this thesis, we focus on fast and parallel of stream clustering using Hadoop Online Prototype. We implemented two state-of-the-art algorithms that aim to efficient cluster streaming data. The challenge, has not been to create a new

clustering method, rather to parallelize existing stream clustering solutions in Hadoop Online Prototype API, so that it can be used for large and continuous datasets.

The first algorithm is based on K-Means. Distributes all points to machines and finds for each one the nearest cluster, from an initial random chosen list of centers, then new cluster centers are created upon this assignment. This step is repeated until a maximum number of iterations is reached or the cluster are so good that no other iteration is needed before the final assignment to the best clusters.

The second algorithm is based on Local Search solving the Facility Location problem as proposed in Guha et al. paper [8]. From a random list of feasible points, we test for each point whether or not the cost we would save , if this point is finally added to the cluster list, is positive. In case a feasible point is gainful to be added in cluster list, algorithm runs a second stage which assigns all points to the clusters in the current cluster list.

This thesis is organized as follows. Chapter 2 describes the background knowledge and related work regarding our work. This includes clustering theory (2.1), K-Means (2.2), Facility Location (2.3), Hadoop framework with Map/Reduce model and Job Submission (2.4), Hadoop Online Prototype (2.5) and the analysis of the algorithms that Guha et al. propose in their paper (2.6). The description of our Map/Reduce algorithm of K-Means lies in Chapter 3 and the one of Facility Location in Chapter 4. Finally, Chapter 5 contains the experiments for every Map/Reduce algorithm that we conducted in several real datasets, along with interesting results.

Chapter 2

Background and Related Work

2.1 Clustering Theory

Data clustering is a common technique for statistical data analysis, which is used in many fields, including machine learning, data mining, pattern recognition, image analysis and bioinformatics. Clustering is the grouping of similar objects into different groups, or more precisely, the partitioning of a data set into subsets (called clusters), so that the data in each subset (ideally) share some common trait - often proximity according to some defined distance measure.

Clustering algorithms can be classified in several types, including for instance, *Exclusive Clustering, Overlapping Clustering, Hierarchical Clustering, Probabilistic Clustering*. In the Exclusive Clustering data are grouped in an exclusive way, so that if a certain datum belongs to a definite cluster then it could not be included in another cluster. In contrast, overlapping clustering, uses fuzzy sets to cluster data, so that each point may belong to two or more clusters with different degrees of membership. In this case, data is associated with appropriate membership values in [0,1] for each cluster. Hierarchical clustering creates a hierarchy of clusters which may be represented in a tree structure called a dendrogram. The root of the tree consists of a single cluster containing all observations, and the leaves correspond to individual observations. Algorithms for hierarchical clustering are generally either agglomerative, in which one starts at the leaves and successively merges clusters together; or divisive, in which one starts at the root and recursively splits the clusters. So, in both cases, after a few iterations the algorithm reaches the final clusters wanted. Finally, Probabilistic clustering uses a completely probabilistic approach. The most widely used probabilistic clustering method is the one based on learning a *Mixture of Gaussians*, where we can actually consider clusters as Gaussian distributions centered on their barycenters. This algorithm is a model-based approach, which consists in using certain models for clusters and attempting to

optimize the fit between the data and the model. In general, each cluster can be mathematically represented by a parametric distribution, like a Gaussian (continuous) or a Poisson (discrete). The entire data set is therefore modelled by a mixture of these distributions. An individual distribution used to model a specific cluster is often referred to as a component distribution.

In this thesis we focus on K-Means and Facility Location algorithms, both of which belong in the category of exclusive clustering algorithms.

2.2 K-Means

J. MacQueen developed an algorithm in 1967 [11] describing a process for partitioning an N-dimensional population into k sets on the basis of a sample. This algorithm is widely known as k-means and appears to give partitions which are reasonably efficient in the sense of minimizing variance. K-means is one of the simplest unsupervised learning algorithms that solve the well known clustering problem. The procedure follows a simple and easy way to classify a given data set through a certain number of clusters k, that is fixed a priori. The main idea is to define k centroids, one for each cluster. A centroid is the "average" (arithmetic mean) of all cluster points. These centroids should initially be placed in a clever way, since their location can affect the result. The better choice is to place them as far away from each other as possible. The next step is to take each point belonging to a given data set and associate it with the nearest centroid. When no point is pending, the first step is completed and an initial grouping is done. At this point we need to re-calculate k new centroids as barycenters of the clusters resulting from the previous step. These new centroids should be the centers of mass of the points assigned to each cluster from the previous step. After we have these k new centroids, a new binding has to be done between the same data set points and the nearest new centroid. With this reassignment a loop has been generated. As a result of this loop we may notice that the k centroids change their location step by step until no more changes are done. In other words centroids do not move any more. Finally, this algorithm aims at minimizing an objective function, in this case a squared error function. The objective function

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} \|x_i^{(j)} - c_j\|^2,$$

where $\|x_i^{(j)} - c_j\|^2$ is the Euclidean square distance between a data point $x_i^{(j)}$ and the cluster center c_j, is an indicator of the n data points from their respective cluster centers. Although it can be proved that the procedure will always terminate, the k-means algorithm does not necessarily find the optimal configuration, corresponding to the global objective function minimum. The algorithm is also significantly sensitive to the initial randomly selected cluster centers. The k-means algorithm can be run multiple times to reduce this effect.

	K-Means Algortihm
Input:	$\mathrm{E} = \{e_1, e_2, .., e_n\}$
	k (number of clusters)
	$maxIter$ (limit of iterations)
Output:	$\mathrm{C} = \{c_1, c_2, .., c_k\}$
	$\mathrm{L} = \{l(e) \mid e = 1, 2, ..., n\}$ (set of cluster labels of E)

Algorithm 2.2.1: K-MENAS$(E, k, maxIter)$

for each $\mathrm{c}_i \in C$ // Random selection
do $\{\mathrm{c}_i \leftarrow \mathrm{e}_j \in E$
for each $\mathrm{e}_i \in E$
do $\begin{cases} \mathrm{l}(\mathrm{e}_i) \leftarrow \text{ARGMINDISTANCE}(\mathrm{e}_i, c_j) & // \ \mathrm{j} \leq k \end{cases}$

$changed \leftarrow false$;
$iter \leftarrow 0$;

repeat
for each $\mathrm{c}_i \in C$
do $\{$UPDATECLUSTER(c_i);
for each $\mathrm{e}_i \in E$
do $\begin{cases} minDist \leftarrow \text{ARGMINDISTANCE}(\mathrm{e}_i, c_j) & // \ \mathrm{j} \leq k \\ \textbf{if } minDist \mathrm{neq} l(\mathrm{e}_i) \\ \quad \textbf{then} \begin{cases} l(\mathrm{e}_i) \leftarrow minDist; \\ changed \leftarrow true; \end{cases} \end{cases}$
until $changed = true$ *and* $iter \leq$ maxIter

2.3 Facility Location

The main problem of clustering is the choice of the number of expected clusters in a dataset. In the Facility Location problem, this is avoided as there is no a priori number of clusters, instead we find all those that minimize the total cost. In the *Facility Location* problem the basic issue is to distribute a number of facilities in a given set of data points, in a way of minimizing the total opening and maintenance cost of the facilities.

In each real-life facility location problem that we study, there is a set of locations at which we may build a facility (such as a warehouse), where the cost of building this facility is z. Furthermore, there is a set of client locations (such as stores) that need to be serviced by a facility, and if a client at location j (c_j) is assigned to a facility at location i (f_i), it incurs a cost that is proportional to the distance between f_i and c_j.

The objective in any case is to determine a set of locations at which to open

facilities so as to minimize the total facility and assignment costs.

$$FC(N,F) = z|F| + \sum_{i=0}^{k}\sum_{j=0}^{N} d(f_i, c_j),$$

where N is the given set of data points, $F = \{f_1, f_2, ..., f_k\} \subset N$ is the set of k cluster centers, z is a parameter for the facility opening cost (assuming uniform opening costs) and d is a distance function $d : N \times N \to \Re^+$. Moreover, for any choice of F defines a set of points $N_i \subseteq N$ that are closer to f_i, with the union of $N_1, N_2, ..., N_k$ being the whole set N.

There are two variants of facility location problem: the *uncapacitated* and the *capacitated* case. In the *uncapacitated* case, each facility can service an unlimited number of clients without restrictions , whereas in the *capacitated* case, each facility can serve, for example, at most u clients.

2.4 Hadoop

Hadoop is the Apache Software Foundation top-level project [1]. Hadoop is the answer to the needs of numerous computations that process a large amount of data to be completed in a reasonable amount of time. The Hadoop open-source project, supplies a powerful framework for the development of highly scalable distributed computing applications. Hadoop allows the developer to focus on the application logic leaving apart the processing details, as the framework is fully responsible for balanced distribution of data and processing to the nodes in a cluster, handling possible failures and other similar issues. This programming model that Hadoop uses is called MapReduce. MapReduce is a distributed data processing model and execution environment that runs on large clusters of commodity machines.

Hadoop includes a number of subprojects :

- Common: The common utilities that support the other Hadoop projects.
- HDFS: A distributed file system that provides high throughput access to application data.
- ZooKeeper: A distributed, highly availably coordination service. ZooKeeper provides primitives such as distributed locks that can be used for building distribuuted applications.
- Avro: A data serialization system for efficient, cross-language RPC, and persistent data storage.
- Chukwa: A distributed data collection and analysis system. Chukwa runs collectors that store data in HDFS, and it uses MapReduce to produce reports.

- HBase: A distributed column-oriented database. HBase uses HDFS for its underlying storage, and support both batch-style computations using MapReduce and point queries (random reads).
- Hive: A distributed data warehouse.Hive manages data stored in HDFS and provides a query language based on SQL, (and which is translated by the runtime engine to MapReduce jobs) for querying the data.
- Mahout: A Scalable machine learning and data mining library, implementing algorithms for clustering, classification and batch based collaborative filtering.
- Pig: A dataflow language and execution environment for exploring very large datasets. Pig runs on HDFS and MapReduce clusters.

Hadoop also provides its own set of data types that are optimized for network serialization and correspond to the known Java built-in data types. Of course, the user can define custom data types if necessary. The data types that are used as keys need to implement the WritableComparable and the data types that are used as values need to implement the Writable interface, which is a subset of WritableComparable. The Writable interface implements the methods that are used for serialization and deserialization of the objects and the WritableComparable implements additionally the methods that are used for the comparison of the keys.

The most common Hadoop data types are:

- Text: equivalent to String.
- IntWritable: equivalent to Integer.
- LongWritable: equivalent to Long.
- FloatWritable: equivalent to Float.
- DoubleWritable: equvalent to Double.
- BooleanWritable: equivalent to Boolean.

2.4.1 Map/Reduce

MapReduce [7] is the programming model implemented by Hadoop for prossesing large datasets that are produced by many different real-world tasks. MapReduce works by breaking processing into two phases: the map and the reduce phase with key/value pairs as input and output; allowing the user-programmer to implement those two functions (map - reduce) to perform the computations needed. First, the MapReduce library splits the input data into M parts and distributes them in parallel to different machines, as shown in Figure 2.1.

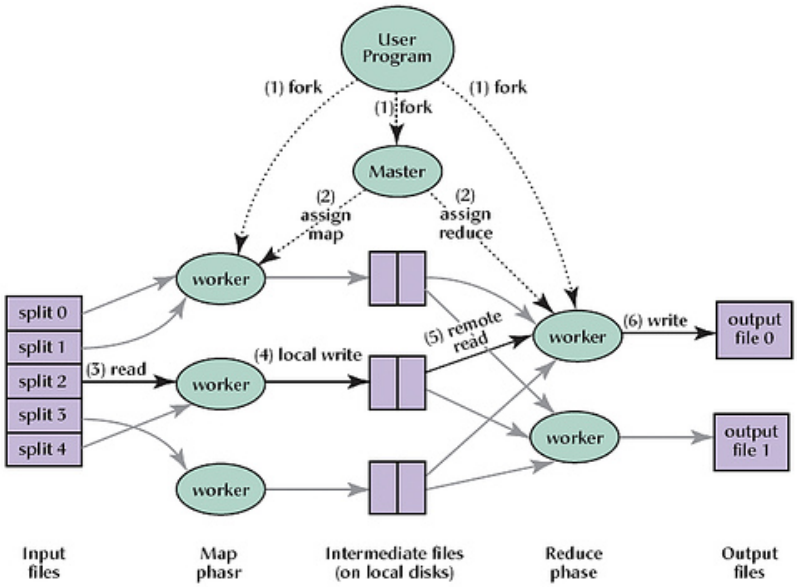

Figure 2.1: MapReduce execution overview.

The framework will convert each record of input into a key/value pair. Each pair will be input to the map function, which is the initial ingestion and transformation step. The map output, which is also a set of key/value pairs (intermediate key/value pairs), is grouped and sorted by key. The result of grouping and sorting is the input of the reduce function.

map	(k1,v1)	→ list(k2,v2)
reduce	(k2,list(v2))	→ list(v2)

The reduce function is applied to each key, in sort sequence, with the key and the set of values that share that key. The reduce method may output an arbitrary number of key/value pairs, which are written to the output files in the job output directory. If the reduce output keys are unchanged from the reduce input keys, the final output will be sorted.

Programs written in this functional style are automatically parallelized and executed on large clusters of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the programs execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

For example, consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar

to the following pseudocode.

Algorithm 2.4.1: MAP($key, value$)

// key is a document name

// value is the document's contents

for each $word \in value$
  do $EmitIntermediate(word, 1)$;

Algorithm 2.4.2: REDUCE($key, IteratorValues$)

// key is a word

// value is a list of values

$result \leftarrow 0$
for each $v \in values$
  do $\begin{cases} result+ = \text{PARSEINT}(v) \\ \text{EMIT}(\text{ASSTRING}(result)) \end{cases}$

A graphical presentation of our example is shown in Figure 2.2 (below) clarifying how our initial data is split and used by *Mappers* and the way mapper outputs are shuffled and sorted so *Reducers* are able to use them to emit the final results.

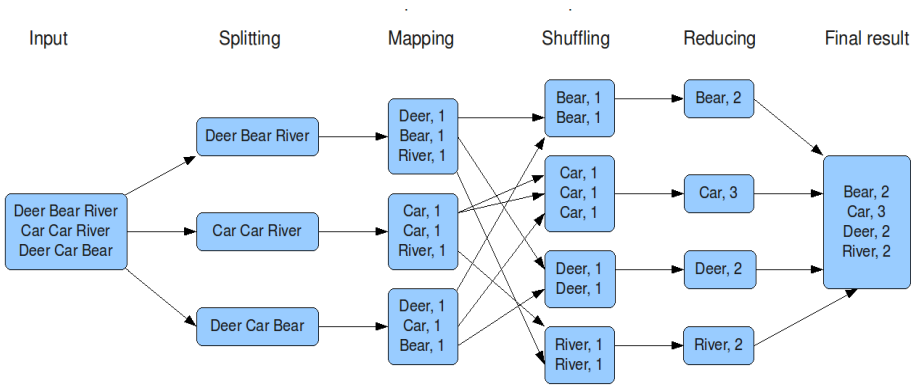

Figure 2.2: Map Reduce data flow for word count example.

2.4.2 Job Submission

The Hadoop framework runs a *MapReduce Job* by dividing it into two types of tasks: *map tasks* and *reduce tasks*. Two processes are responsible for the

execution of MapReduce Jobs a *jobTracker* and a number of *TaskTrackers*. The *TaskTrackers* manage the execution of individual map and reduce tasks on a compute node in the cluster. The *jobTracker* accepts job submission, provides job monitoring and control, and manages the distribution of tasks to the TaskTracker nodes. TaskTrackers periodically send a heartbeat signal to jobTracker. This heartbeat informs the jobTracker that the task is still alive and also a part of heartbeat indicates whether the TaskTracker is ready to run a new task. When a MapReduce job is submitted, the user has to provide the system with a series of necessary parameters regarding the job. Such parameters included, the input and output type format and destination in HDFS, the classes of map and reduce functions, the JAR file(s) that contain the map and reduce functions and maybe other support classes, as well any possible configuration informations needed. The first action taken by the framework, once a new job is submitted, is to divide the input into fixed-size pieces (usually same size with HDFS block - 64MB by default) called *input splits* and create one map task for each split. Each map task calls the user-defined map function once per record (one line) in the split producing an intermediate key/value pair and writes it locally. At this point, Hadoop guarantees that the input to every reducer is sorted by key. The process by which the system performs the sort and transfers the map outputs to the reducers as inputs, is known as the *shuffle*. The shuffle is the heart of MapReduce, and is where the "magic" happens. So, the reduce task needs the map output for its particular partition from several map task across the cluster. The map tasks may finish at different times, so the reduce task starts copying their outputs as soon as each completes. This is known as the *copy phase* of the reduce task. When all map outputs have been copied, the reduce task moves into the *merge phase*, which merges the map outputs, maintaining their sort ordering. The last phase is the *reduce phase*. During the reduce phase the reduce function is invoked for each key in the sorted output. The output of this phase is written directly to the output distributed filesystem (HDFS). The above process is depicted in Figure 2.3

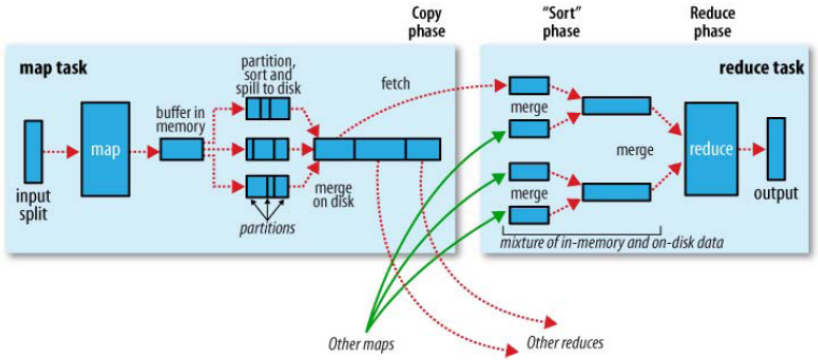

Figure 2.3: Shuffle and Sort in MapReduce.

2.5 Hadoop Online Prototype

MapReduce is a framework for processing huge datasets on certain kinds of parallelizable problems with primary focus on time of job completion. As analyzed in Section 2.4 the batch-processing implementation of MapReduce, is based on the materialization of map and reduce stages, for reliability and storage stability. However, this strategy has a high probability of slowdowns or failures at worker nodes. In attempting to resolve these issues, Condie et al. [6] proposed a modified MapReduce architecture in which, along with some other additional features, intermediate data is *pipelined* between tasks, saving valuable time in the execution process. This pipelined architecture of Hadoop called *Hadoop Online Prototype* (HOP).

The main advantage of HOP to the unmodified Hadoop is that reducers are able to begin consuming the output produced by mappers after it is produced without requiring all mapper executions to complete, and without writing it to HDFS.

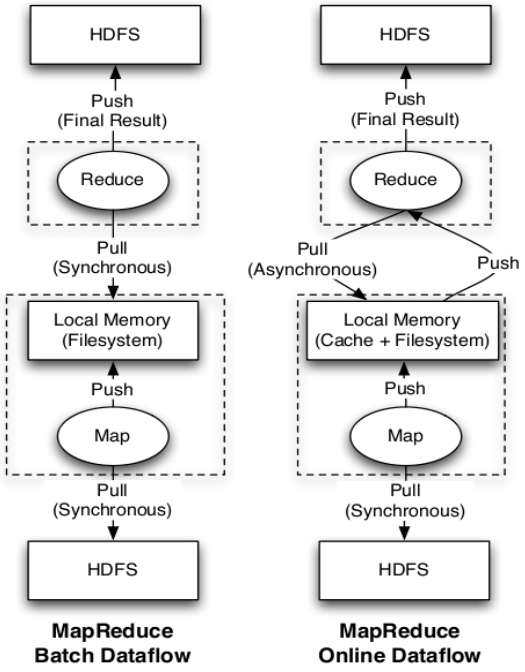

Figure 2.4: Hadoop dataflow for batch (left) and pipelined (right) processing of MapReduce computations.

Figure 2.4 depicts the dataflow of two MapReduce implementations with this basic difference. The dataflow on the left corresponds to the output materialization approach used by stock Hadoop; the dataflow on the right allows pipelining with the addition of a "flush" function, which pushes data from the map task

to the reduce tasks. This *pipelining* mechanism can be applied intra-job (map tasks push their output to reducers as it is produced) and inter-job (reducer tasks of a previous job pipeline their output to the map tasks of a next job). By its design HOP [2] is able to handle two important types of functionality: *online aggregation* and *continuous queries*. *Online aggregation* [9] is a technique that allows the user to have an approximation of the final output during the course of execution. Traditional MapReduce implementations suffer in interactive data analysis, by losing valuable time expecting the job to be completely executed in order to produce the final results. In HOP, reducers begin processing data as soon as it is produced by mappers, so they can generate, according to the percentage of input data, a "quick and dirty" approximation over the final output. This approximation is called *"snapshot"* and this mechanism can be used at various intervals based on the job progress after system configuration. In this manner, time for data analysis can be reduced by several orders of magnitude. *Continuous queries* is the second new functionality supported by HOP. With this feature continuous running jobs can accept data as it becomes available and analyze it immediately. This *Continuous queries* functionality is critical for the analysis of streaming data.

2.6 Stream Clustering Algorithms

In computer science, streaming algorithms are algorithms for processing data streams in which the input is presented as a sequence of items and can be examined in only a few passes (typically just one). These algorithms have limited memory available to them (much less than the input size) and also limited processing time per item. In our thesis, we focus on the stream clustering algorithms that Guha et al. [8] propose in their paper. Guha et al. begin their analysis with a simple algorithm, and after a row of improvements that we analyze below in this section, they end up with a final Local Search algorithm solving Facility Location problem. The one of the algorithms we have implemented is this final algorithm. Finally, we have also implemented a second algorithm, which is a simple variant of K-Means algorithm.

2.6.1 Guha et al. Algorithms

Baseline Strategies

Guha et al. investigate algorithms that examine the data in a piecemeal fashion. In particular, they begin their study with the performance of a simple divide-and-conquer algorithm, called Small-Space. This algorithm divides the data into l pieces, clusters each of these pieces, and then again clusters the centers obtained (where each center is weighted by the number of points assigned to it).The l parameter will be set so that both dataset (S) and lk centers fit in the memory. The next algorithm they propose (Smaller-Space) is similar to the previous piecemeal approach except that, instead of reclustering only once,

it repeatedly reclusters weighted centers. For this algorithm, they prove that, if it reclusters a constant number of times, a constant-factor approximation is still obtained, although, as expected, the constant factor worsens with each successive reclustering. However, the Smaller-Space algorithm, has a problem since the intermediate medians of each clustered piece must be stored in memory, this implies that the number of subsets that the original dataset is partitioned into is limited by memory capacity.

Figure (2.5) below shows a graphical representation of those two algorithms.

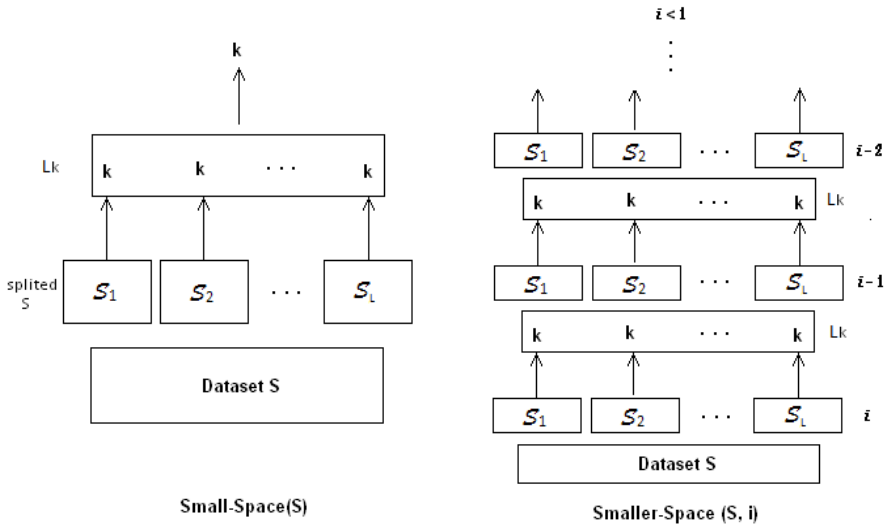

Figure 2.5: Dataflow for Small-Space (left) and Smaller-Space (right) algorithms.

Hierarchical Strategy

The next algorithm of Guha et al. uses a more clever implementation of a hierarchical scheme (Figure 2.6) to get around of the Smaller-Space algorithm problem. The algorithm takes as input the first m points ($m = \sqrt{M}$, where M is the memory size) and reduce them with a bicriterion algorithm to $O(k)$ (say $2k$) points. As usual, the weight of each intermediate median is the number of points assigned to it in the bicriterion clustering. (Assume m is a multiple of $2k$). This is repeated until m intermediate medians are generated.

At this point, these m first-level medians are clustered into $2k$ second-level medians and proceed. In general, it maintains at most m level-i medians, and, on seeing m, generates $2k$ level-$(i+1)$ medians, with the weight of a new median as the sum of the weights of the intermediate assigned to it. Finally, after seeing all the original data points, all the intermediate medians are clustered to k final medians by a primal dual algorithm. The approximation quality of this

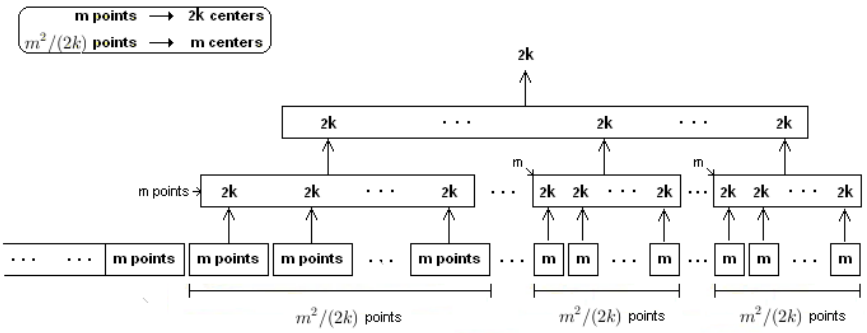

Figure 2.6: Hierarchical scheme for streaming model.

algorithm will depend heavily on the number of levels. Guha et al. use the local search algorithm in [5] to provide a bicriterion approximation in *space linear in* m, the number of points clustered at a time.

Hierarchical Speed Up

***Leaf-Level* :** Then Guha et al. try to reduce the running time of this scheme. The running time of this hierarchical clustering is dominated by the contribution from the first level and the local search they are using is quadratic, so they proposed a sampling-based algorithm that requires $O(nk)$ time, where n denotes the size of the input stream, for the generation of the first level intermediates. Consider the following algorithm:

1. Draw a sample of size s = $\sqrt{nk}$.
2. Find k medians from these s points using a primal dual algorithm.
3. Assign each of the n original points to its closest median.
4. Collect the n/s points with the largest assignment distance.
5. Find k medians from among these n/s points.
6. We have at this point $2k$ medians.

This sampling-based scheme is used to develop a one-pass and $O(nk)$-time algorithm.

1. *Leaf-level point clustering*
   - Input the first $O(M/k)$ points, and use the randomized algorithm above to cluster this to $2k$ intermediate median points.
2. *Intermediate-level point clustering*

- Use a local search algorithm to cluster $O(M)$ intermediate medians of level i to $2k$ medians of level-(i+1).

3. *Final clustering*

   - Use the primal dual algorithm [10] to cluster the final $O(M)$ medians to k medians.

Notice that the algorithm remains one pass since the $O(\log n)$ iterations of the randomized subalgorithm just add to the running time. Thus, over the first phase, the contribution to the running time is $O(nk)$. Over the next level, we have $\frac{nk}{M}$ points, where M is the memory size, and if we cluster $O(M)$ of these at a time taking $O(M^2)$ time, the total time for the second phase is $O(nk)$ again. The contribution from the rest of the levels decreases geometrically so the running time is $O(nk)$.

***Intermediate-Level* :** The second step of the algorithm above is using a quadratic-time local search. They propose that this time can be reduced by relaxing the number of clusters. The parameter k (number of clusters) is a target, and *need not be held fixed in the intermediate stages of the algorithm* and local search allows this flexibility. The ability to relax the parameter k in the intermediate steps of the algorithm provides an interesting contrast to *k-Means* which (as defined commonly) does not relax the number of clusters.

Lagrangian Relaxation techniques provide a powerful tool for combinatorial optimization problems and the Facility Location minimization is a Lagrangian relaxation of the k-median problem. Algorithm, Guha et al. present is a new local search-based algorithm solving the Facility Location Problem in order to speed up the local search step, by relaxing the number of clusters. This solution relies on a simple algorithm of Charikar and Guha [5], referred to as *CG*, for solving facility location on a set N of n points in a metric space with metric (relaxed metric) $d(.,.)$, when the facility cost is z. We briefly describe *CG* below.

Assume that we have a feasible solution to facility location on N given $d(.,.)$ and z. That is, we have some set $I \subseteq N$ of currently open facilities and an assignment for each point in N to some (not necessarily the closest) open facility. For every $x \in N$, we define *gain* of x to be the cost we would save (or further expend) if we were to open a facility at x (if one does not already exist), and then perform all possible advantageous reassignments and facility closings, subject to the following two constraints: First, points cannot be reassigned except to x and, second, a facility can be closed only if all its members are first reassigned to x.

Algorithm CG(data set N, facility cost z)

1. Obtain an initial solution (I, f) ($I \subseteq N$ of facilities, f an assignment function) that gives a n-approximation to facility location on N with facility cost z.

2. Repeat $\Omega(\log n)$ times:

- Randomly order N.
- For each x in this random order: calculate $gain(x)$, and if $gain(x) > 0$, add a facility there and perform the allowed reassignments and closures.

***CG Speed Up* :** Their final algorithm, uses the CG algorithm as a subroutine. But, the CG algorithms running time $\Theta(n^2 \log n)$ is expensive for large data streams. Therefore, they describe a new local search algorithm, referred to as FL, that relies on the correctness of the above algorithm, but avoids the quadratic running time by taking advantage of the structure of local search and the sampling-based scheme that discussed earlier. Guha et al. prove that instead of evaluating $gain$ for every point x, it would be still likely to choose good medians from a randomly choosen set of $\Theta(\frac{1}{p} \log k)$ points and finish the computation sooner.

Algorithm FL(N**,** $d(.,.)$**,** z**,** ϵ**,** (I, α) **)**

1. Begin with (I, α) as the current solution.
2. Let C be the cost of the current solution on N.
   Consider the feasible centers in random order, and, for each feasible center y, if $gain(y) > 0$, perform all advantageous closures and reassignments (as per $gain$ description), to obtain a new solution (I', α'). [α' should assign each point to its closest center in I' .]
3. Let C' be the cost of the new solution; if $C' \leq (1 - \epsilon)C$, return to Step 2.

Complete Algorithm

Their final algorithm, the one we have implemented, uses the FL algorithm, as a subroutine, also using a binary search for the best selection of the parameter *zeta*. The algorithm begins with an initial solution and an initial range for the facility cost *zeta* (between 0 and an easy-to-calculate upper bound); then a binary search is performed within this range to find the appropriate value of *zeta* that gives the right number k of facilities needed (clusters). This search, for the parameter *zeta*, calls the FL algorithm repeatedly (iterative step), with a parameter ϵ that controls convergence, for a number of feasible points. Then it checks if the total cost changes very little from one iteration to the next and the solution is far from k centers, then we have gotten the value of *zeta* incorrect.

The final *k-Median* algorithm for a data set N with distance function d that uses FL as a subroutine:

Algorithm LSEARCH(N, $d(.,.)$, k, ϵ, ϵ')

1. $z_{min} \leftarrow 0$.
2. $z_{max} \leftarrow \sum_{x \in N} d(x, x_0)$ (for x_0 an arbitrary point in N).
3. $z \leftarrow (z_{min} + z_{max})/2$.
4. $(I, \alpha) \leftarrow$ InitialSolution(N, z)
5. Randomly pick $\Theta(\frac{1}{p} \log k)$ points as feasible medians.
6. While medians $\neq k$ and $z_{min} < (1 - \epsilon')z_{max}$:
   - Let (F, g) be the current solution.
   - Run $FL(N, d, \epsilon, (F, g))$ to obtain a new solution (F', g')
   - If $k \leq |F'| \leq 2k$, then exit loop.
   - If $|F'| > 2k$, then $z_{min} \leftarrow z$ and $z \leftarrow (z_{max} + z_{min})/2$;
   - else if $|F'| < k$, then $z_{max} \leftarrow z$ and $z \leftarrow (z_{max} + z_{min})/2$;
7. Return our Solution (F', g').

2.6.2 Streaming K-Means

Finally, we devised and implemented an other stream algorithm to handle the continuous data. The idea is based on an appending-scheme, as shown in the Figure 2.7 (below). The output of each data set is appended in the next coming data set and is treated as a normal part of the whole file. We implemented this scheme in *K-Means* algorithm.

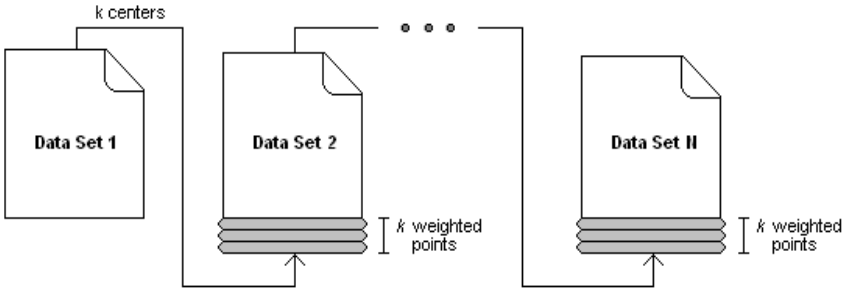

Figure 2.7: Appending scheme for streaming model.

Chapter 3

K-Means Algorithm

3.1 Introduction

As we discussed in Section 2.2 K-Means is one of the simplest algorithms for solving the clustering problem. K-means clustering is a method of cluster analysis which aims to partition n data points into k clusters in which each data point belongs to the cluster with the nearest mean. Thus, k-means main goal is to minimize the within-cluster sum-of-squared distances.

Suppose that we have a set of data points $(x_1, x_2, ..., x_n)$, where each point is a d-dimensional vector. After the assignment to k centroids $(\mu_1, \mu_2, ..., \mu_k)$ we will have k partitions $S = \{S_1, S_2, ..., S_k\}$.

$$\underset{S}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x_j \in S_i} \|x_j - \mu_i\|^2$$

As mentioned in Section 2.2 our first stream clustering algorithm relies on the K-Means algorithm, which has already been implemented in conventional Hadoop in the Mahout project. Our implementation has similar logic with K-Means clustering that Mahout implements[1]. The key difference is that we implemented K-Means in HOP considering the "snapshot" mechanism that it provides. So, we allow the user to see an approximation of the final results early on before the job is completed. Also, as we discussed in subsection 2.6.2, our K-Means implementation is able to receive a continuous series of data appending each time the final clusters to the next dataset as weighted points (where weight is the number of the assigned points).

[1] Once we completed our implementation, we found another paper implementing K-Means in Hadoop [12] with a similar way Mahout does.

3.2 Description of the Algorithm

Firstly, we implemented a streaming algorithm based on the appending scheme that discussed earlier in Section 2.6.2. According to this scheme each dataset returns K centers, which are appended at the next dataset as weighted points, with weight the number of points assigned to this weighted point. The algorithm we used to cluster each datasets is *K-Means*.

Our K-means implementation uses one Map and one Reduce jobs called in two different points. Firstly, Map/Reduce job with the knowledge of all cluster centers, finds, for each given point, the closest cluster center and assigns it to this cluster; then, new cluster centers are recomputed with the assignment that results. Figure 3.1 depicts the above logic.

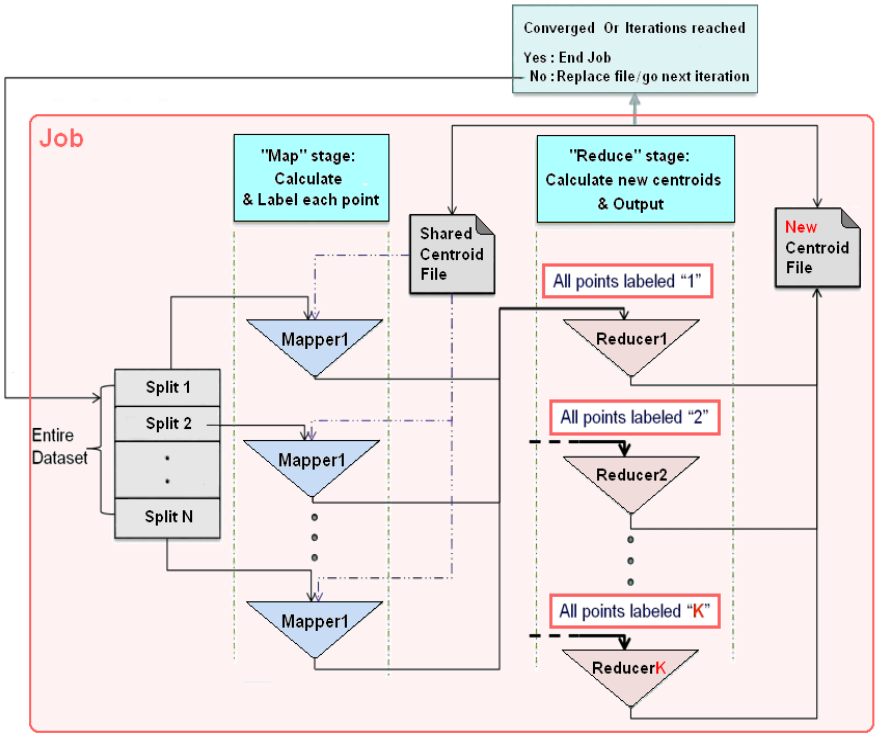

Figure 3.1: Iterative Map/Reduce Job.

This job is iterative under two conditions: The first condition is that the number of iterations should not exceed a given "max-iterations" parameter, The second condition checks for convergence, which means that our algorithm converges and breaks the iterative part if the distance of all new cluster centers from those in the prior iteration, one by one, do not exceed, distance, the given parameter about convergence. The second Map/Reduce job is called, when

iterations are completed and computes simple assignment of all points to the last found cluster centers, by the iterative part.

The driver routine that the algorithm calls the Map/Reduce Job is shown in Figure 3.2 (below)

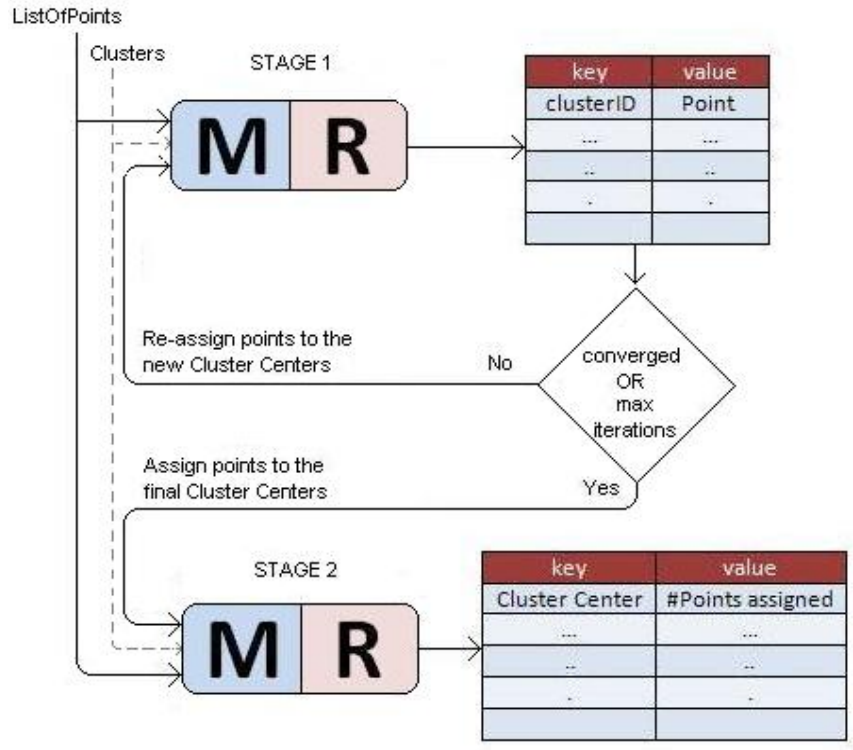

Figure 3.2: Driver routine for K-Means Algorithm.

In the remainder of this chapter, we provide the description of the Map and Reduce functions of the algorithm along with their pseudocode, also we analyze the detailed description of two custom Writable variables, Cluster and KMeansInfo, that we used. In the map/reduce functions described below, we give the input and output types of the variables and also an example of the key/value pairs that are read and written. The notation that is used in the following sections to express the key/value pairs of the map/reduce framework uses underlining for keys and parenthesis for values, i.e., key, (value).

3.2.1 Custom Writable Variables

For better understanding of the Map/Reduce stages, in this section we analyze two *Custom Writable Variables* that we created in order to keep some necessary information between Tasks and Jobs. As discussed in Section 2.4, Hadoop allows

users to define their own custom variables if necessary. The first such variable is the *Cluster Writable* that keeps all the information needed for a cluster and the points assigned to it. The second is the *KMeansInfo Writable* that keeps all the information needed by reducers about the points and the clusters they are assigned to.

Cluster Writable

Our need to categorize clusters based on their center and their assigned points, and also recompute new centers and thus new clusters, led us to implement this custom Writable variable, which we called "*Cluster*". This variable is created in order to help us emit composite values and is used to express clusters. It contains six fields: clusterId, center, numPoints, pointTotal , sqrTotal, converged. clusterId is the unique identity of a cluster, center is a vector of doubles expressing the dimensions of the cluster center. This attribute is very important for us, as we use it in map functions for the assignment of all point to the nearest center of cluster and in reduce functions for the convergence check, the distance between new recomputed and prior center, numPoints is an integer exressing the number of points this cluster is consisted of and is used in reduce task of the first stage to recompute the center , pointTotal is a vector of doubles expressing the sum per dimension of all points assigned in cluster and it is needed in reducers of the first stage when recomputing the center (each dimension is divided by the numPoints of the Cluster), sqrTotal is a vector of doubles expressing the sum per squared dimension of all points assigned to cluster center and with pointTotal is used in the end of clustering process to check the the efficiency of the final clustering. and converged is a boolean shows, whether the previous cluster center is farther from current (recomputed) center than given parameter of convergence, or not.

Supposing that we have two points, let us say *p1*:{1,1} and *p2*:{2,1} assigned to the same *cluster center*:{2,2} then the *Cluster* attributes would be: clusterId = 1 (random choice but different from other existing objects of *Cluster*), center = {2,2}, numPoints = 2, pointTotal = {3 ($p1_x$+$p2_x$) , 2 ($p1_y$+$p2_y$)}, sqrDistance = {5 ($p1_x^2 + p2_x^2$) , 2 ($p1_y^2 + p2_y^2$)}, converged = false (false is the initialization of converged as no previous step has been made to check convergence).

Input Data	Cluster					
point	*id*	*center*	*num*	*pntTot*	*sqrTot*	*converge*
1 1	1	[2 , 2]	2	[3 , 2]	[5 , 2]	false
2 1						

In the Cluster column there is only one record as we assume that both points assigned to the same cluster, so they were grouped by cluster key.

The class *Cluster* implements the following methods:

- associated constructors: All Writable implementations must have a default constructor so that the MapReduce framework can instantiate them.

- setters and getters for the variables.
- *toString*(): returns the String representation of the variable. It is called to write the reducers output to the HDFS filesystem, if the output format is TextOutputFormat.
- *readFields*(): deserializes the bytes from the input stream by delegating to each object. It is called by a mapper to read from the HDFS.
- *write*(): serializes each object in turn to the output stream. It is called by the collect() function and writes the variable to the HDFS, if the output format is SequenceFileOutputFormat.
- *getIdentifier*(): returns a V or a C, depending on cluster center convergence, concatenated with the cluster id; so we can easily check which of the clusters have converged.

KMeansInfo Writable

In order to define a cluster we need to know how many and which specific points comprise this cluster. This need to keep points with their multiplicity and pass this information between tasks, led us to implement a custom Writable variable, which is called "*KMeansInfo*". This variable is created in order to help us emit composite values and is used to express the main attribute of a cluster,i.e., its points with their multiplicities. This composite variable is needed to pass the appropriate information needed from Mappers to Reducers. It contains 4 fields: numPoints, point, pointsTotal, pointSqrTot. numPoints is an integer expressing the multitude of points collected at this point (weighted points) this is used in mapper's output when the income data contains weighted points with multiplicity greater than one, so reducers have the knowledge of how many points are assigned to each cluster. The variable point is a vector of doubles expressing the dimensions of our point and is used in reducers to compute the new center, by summing all points per dimension assigned to a cluster and dividing with the number of points assigned. The third attribute, pointsTotal, is a vector of doubles that expresses the point (weighted or not) that is to be assigned to a cluster in map tasks. If a point has multiplicity "one" (numPoints=1) then the pointsTotal vector is the same with the point vector, else each dimension is the multiplied, with numPoints variable, dimension of point. sqrTotal is a vector of doubles, each dimension expressing the squared dimension of point.

To make this clearer we give an example showing what *KMeansInfo variable* would be created with two types of data. The one has multiplicity greater than one while the other has not. If points have some multiplicity then next to their dimensions we have an "m" character that separates the point from multiplicity. It is also possible to have this "m" character while multiplicity is one and the number next to "m" is 1:

Input Data			KMeansInfo				
point	*with*	*multiplicity*	*numPnts*	*point*	*pntTot*	*pntSqrTot*	
3	*2*	*m1*	1	[3 , 2]	[3 , 2]	[9 , 4]	- Simple
2	*1*	*m2*	2	[2 , 1]	[4 , 2]	[8 , 2]	- Weighted

The class *KMeansInfo* implements the following methods:

- associated constructors: All Writable implementations must have a default constructor so that the MapReduce framework can instantiate them.
- setters and getters for the variables.
- *toString*(): returns the String representation of the variable. It is called to write the reducers output to the HDFS filesystem, if the output format is TextOutputFormat.
- *readFields*(): deserializes the bytes from the input stream by delegating to each object. It is called by a mapper to read from the HDFS.
- *write*(): serializes each object in turn to the output stream. It is called by the collect() function and writes the variable to the HDFS, if the output format is SequenceFileOutputFormat.

3.2.2 K-Means in Map/Reduce

As we mentioned before the first step of our algorithm is executed iteratively. In each iteration, the algorithm refines the cluster centers by recomputing new centroids upon the points assigned to each center. This assignment is based on Euclidean distance, so each point is assigned to the nearest center. Iterations stop when the distance of all new centers, one-by-one, to the previous centers is smaller than a factor delta, where delta is the convergence input (our third input), from those before re-computation or when the number of iterations reaches a higher bound which is our second input in algorithm (referring to max iterations).

The first iteration begins after a random initialization of cluster centers, selected from the data set that is given as input. Obviously the computed clusters in each iteration form the input to the next iteration.

Map

The **Mapper** interface is a generic type, with four type parameters that specify the input key, input value, output key, and output value types of the map function. For this Map function, the input key is a long integer offset, the input value is a line of text containing one point. The Map function's mission is to find for each point the nearest cluster center. Each mapper knows all current cluster centers (a global list) and processes one point at a time. As shown in the algorithm KMeansMapper below, a mapper searches the minimum

distance between the processing point and the centers in the known cluster list and returns the id of the nearest cluster as the output key and a KMeansInfo-type variable, which practically contains the current point, as the output value. KMeansInfo is the custom writable variable that was described in the previous section, and is need to keep track of informations for each processed point beyond its dimensions, such as the point's weight and some other information for the ensuing computation of the Sum of Square Distances for each cluster.

InputTypes : LongWritable , (Text)

OutputTypes : Text , (KMeansInfo)

The input of this job is a file containing a set of data points, each mapper takes as input one line at a time that corresponds to a data point, and a list of clusters. At the process of the very first set of data points, mappers read data of type "*x y m1*", because all points have multiplicity "1" (one) . In the next iterations, where we have appended the output of a previous execution (streaming model for K-Means, in section 2.6), the mappers apart from the previous format, will also read k data points with further information. This k is because k centers have been computed from previous inputs and thus k weighted points have been appended to this file. The extra information includes the weight (number of points assigned) of each point (cluster center), and additional information about the data points assigned to the center. An example of such an input is shown in the end of this section. So, once a mapper reads a line, splits it to its components. One of this components is the data point that mapper uses to find the minimum distance from the centers in cluster list. When the nearest cluster center is found, the mapper creates a KMeansInfo variable with the appropriate data and emits the id of nearest cluster as the output key and this KMeansInfo variable as the output value. Key output is also concatenated with a random number between 1 and the number of working machines divided by the size of cluster list. So, we distribute the intermediate data to more than one machine for each cluster increasing the scalability of our algorithm. The pseudocode below depicts the logic of our mapper.

Algorithm 3.2.1: KMeansMapper($key, value$)

global $listOfClusters$
global $reducers$
local $point, multi, pointTot, sqrTot$
local $nearestDistance, nearestCenter$
local $keyOutput$
local $kmInfo$

$point, multi, pointTot, sqrTot \leftarrow$ SPLIT($value$)

$nearestDistance \leftarrow MaxValue$
$nearestCenter \leftarrow null$

for each $cluster \in listOfClusters$
do {
  $center \leftarrow$ GETCENTER($cluster$)
  $distance \leftarrow$ DISTANCE($center, point$)
  if $distance < nearestDistance$
  then {
    $nearestDistance \leftarrow distance$
    $nearestCluster \leftarrow cluster$

$random \leftarrow$ RAND($reducers/(listOfClusters.size() + 1)$)
$keyOutput \leftarrow id + random$
$kmInfo \leftarrow (multi, point, pointTot, sqrTot)$

output ($keyOutput, kmInfo$)

Example : If we suppose that we have from initialization the cluster centers: [1 , 2],[1 , 5],[5 , 3], with ids 1, 2, 3 respectively, and a set of data points in the input file, as shown on the left side of the arrows below, then, after the map execution, the output key for each point, would be the id of the nearest cluster found and the output value would be a KMeansInfo variable as shown below. In the example below we have as input 4 simple and 2 composite points. As discussed earlier, we understand that the composite points have been the result of appending the previous cluster centers to the current data set. From the simple points it is easy to conduct the KMeansInfo variable, but from the composite may not be so obvious. Before discussing the example below, we should mention that in composite input, the reducers from previous step have put all the information needed about clusters in a text line, separated with a character "m",i.e., cluster point $\boldsymbol{m}$ number of assigned points to cluster $\boldsymbol{m}$ vector with sum of points assigned to cluster per dimension $\boldsymbol{m}$ vector with sum of points assigned to cluster per squared dimension.

$$\begin{array}{llll} & & id \quad num \quad point \qquad Total \qquad SqrTot \\ 2\ \ 6\ \ m1 \rightarrow & \underline{2r}\ , & (\ \ 1\ \ [\ \ 2\ \ ,\ \ 6\ \ ]\ [\ \ 2\ \ ,\ \ 6\ \ ]\ [\ \ 4\ \ ,\ 36\ \ ]) & \end{array}$$

$$
\begin{aligned}
1\ \ 5\ \ m1 \rightarrow\ \ & \underline{2r}\ ,\ (\ 1\ [\ 1\ ,\ 5\ ]\ [\ 1\ ,\ 5\ ]\ [\ 1\ ,\ 25\ ]) \\
5\ \ 3\ \ m1 \rightarrow\ \ & \underline{3r}\ ,\ (\ 1\ [\ 5\ ,\ 3\ ]\ [\ 5\ ,\ 3\ ]\ [\ 25\ ,\ 9\ ]) \\
1\ \ 2\ \ m1 \rightarrow\ \ & \underline{1r}\ ,\ (\ 1\ [\ 1\ ,\ 2\ ]\ [\ 1\ ,\ 2\ ]\ [\ 1\ ,\ 4\ ]) \\
1\ 1\ m3\ m2\ \ 2\ m10\ \ 2\ \rightarrow\ \ & \underline{1r}\ ,\ (\ 3\ [\ 1\ ,\ 1\ ]\ [\ 2\ ,\ 2\ ]\ [\ 10\ ,\ 2\ ]) \\
5\ 4\ m2\ m10\ 9\ m50\ 41 \rightarrow\ \ & \underline{3r}\ ,\ (\ 2\ [\ 5\ ,\ 4\ ]\ [\ 10\ ,\ 9\ ]\ [\ 50\ ,\ 41\ ])
\end{aligned}
$$

So, if we take the last input data, from the example above, and split it on "m" then we would have 4 elements: {5 4}{2}{10 9}{50 41}. If we consider the first to be the point (previous cluster center), the second to be the weight (number of points assigned), the third element to be the sum of assigned points per dimension, and the last to be the sum of assigned points per squared dimension, then it is much easier to create the KMeansInfo variable as shown above. In composite points we do not have to calculate the pointsTotal and pointSqrTot attributes of KMeansInfo variable, as they are already given. The r character in key output, as we said before, is a random number that helps us distribute our data to more reducers. In this case r can take values between 1 and numberOf Reducers/(k). For example, if our job runs 90 reducers and our k parameter is 3 then each clusters data will be distributed to a maximum of 30 reducers instead of 1.

Reduce

The reducer function is defined similarly with the mapper. We also have four formal type parameters that specify the input key, input value, output key, and output value. The difference here is that we have a restriction: the input key and value parameters must be the same type of the mapper's output key and value parameter types. Our reducers know not only the current cluster centers, as the mappers, but also the delta factor that is used in convergence, since the convergence check is executed in this step. More specifically, as discussed earlier, reducers take as input key a cluster id and as input valueList the points (in KMeansInfo-type) that share the same id.

InputTypes : Text , (KMeansInfo)

OutputTypes : Text , (Cluster)

Reducers take as input the output produced by the mappers. After, the output data of mappers, has been shuffled and sorted, it has been grouped by key. Each reducer takes a unique key with all the values that share the same key. Thus, each reducer reads as input, data of type { **key** , [$\{KMeansInfo_1\}$, $\{KMeansInfo_2\}$,...,$\{KMeansInfo_n\}$]}. The Reducer uses the input key to find the cluster in the given cluster list. Once the cluster is found, the reducer runs on the valueList, which actually is a list of *KMeansInfo* objects and adds all the points on the cluster. This practically means that from each KMeansInfo object, which consists of four variables (multiplicity, point, pointsTotal, sqrTotal),

reducers, after dimensions of each point have been multiplied with the variable of multiplicity, add all points, so weighted points are taken into account. This value, which actually is a vector, is stored in the *pointTotal* attribute of the cluster. Multiplicities are also added, and the final sum is an integer number, expressing the total number of points assigned to the cluster, which is stored in the numPoints attribute. The output parameters of this process includes, the cluster's id (as the output key), and the cluster itself (as the output value).

The reducer logic is shown in the pseudocode below.

Algorithm 3.2.2: KMeansReducer($key, valueList$)

global $listOfClusters$
local $cluster, point, multi$
local $center, totalPoint, sumPoints$
local id
local $kmInfo$

$id \leftarrow$ getId(key)
$cluster \leftarrow$ findCluster($listOfClusters, id$)

while hasNext($valueList$)
do
- $kmInfo \leftarrow$ getNext($valueList$)
- $multi \leftarrow$ getMulti($kmInfo$)
- $point \leftarrow$ getPoint($kmInfo$)
- $sumPoints \leftarrow$ addMulti($cluster, multi$)
- $totalPoint \leftarrow$ addPoint($cluster, point$)

setNumPoints($cluster, sumPoints$)
setPointTotal($cluster, totalPoint$)

output ($id, cluster$)

example : This reduce method takes as input the output of the above map method. So, if we assume as input the output of the map method and a delta factor d = 0.5, then the outcome of the reduce step would be as follows:

1 , (3 [1 , 1] [2 , 2] [10, 2])
1 , (1 [1 , 2] [1 , 2] [1 , 4]) } 1 (1 [1 ,1.25] 4 [3 , 4] [11 ,6] *false*)

2 , (1 [2 , 6] [2 , 6] [4 ,36])
2 , (1 [1 , 5] [1 , 5] [1 ,25]) } 2 (2 [1.5, 5.5] 2 [3 ,11] [5 ,61] *false*)

3 , (1 [5 , 3] [5 , 3] [25, 9])
3 , (2 [5 , 4] [10, 9] [50, 41]) } 3 (3 [5 ,3.66] 3 [15 ,12] [75 ,50] *false*)

In the example above we can see the grouped by key data on the left side and the clusters created on the right side. First of all we see in the clusters the new recomputed centers; before, the cluster with id=3, had {5 3}, as center and now {5 3.66}. This can easily be exported if we multiply the dimensions of each point with its multiplicity and divide the sum per dimension with the sum of points assigned; $(5 * 1 + 5 * 2)/3 = 5$ and $(3 * 1 + 4 * 2)/3 = 3.66$. The vector attributes of each cluster are simply computed by summing the KMeansInfo corresponding vectors per dimension. Finally, we see that none of the clusters has converged, as their last value is false. This last boolean attribute depends on the distance of the new centers to the previous. If this distance is greater than the factor delta then the cluster has not converged. We can see that all new cluster centers are farther than delta:

$$\begin{aligned} distance(\ [\ 1,\ 2\ ], [\ 1,\ 1.25\ ]\ )\ &>\ 0.5 \\ distance(\ [\ 1,\ 5\ ], [\ 1,\ 5.5\ \ ]\ )\ &>\ 0.5 \\ distance(\ [\ 5,\ 3\ ], [\ 5,\ 3.66\ ]\ )\ &>\ 0.5 \end{aligned}$$

Once all reducers have finished, a function outside Hadoop recomputes the new center of each cluster by dividing the *pointTotal* vector with the *numPoints* variable per dimension. Finally, a second function checks whether each cluster has converged or not, by the Euclidean distance of the old and the recomputed cluster center. If the recomputed cluster center is not farther by a *delta* factor (convergence delta, given as a input variable) to the old one, then the cluster has converged and this is stored in the *converged* attribute of the cluster. If all clusters one-by-one have converged then iteration step stops and our algorithm forwards to the next step, the last assignment.

The second step of our algorithm is called when iterations have finished and the final cluster centers had been found. This step makes a last assignment of all points to the final centers. The Map/Reduce job is the same with the one in iterative part. The reason why we repeat it is because we want to make a last assignment to our points, so after this job we can compute the Sum of Square distances of all assigned points to the final centers of each cluster, without any recomputed centers mess with the final result.

Chapter 4

Facility Location Algorithm

4.1 Introduction

As we discussed in Section 2.3 Facility Location, also known as location analysis, is a branch of operations research concerning itself with mathematical modeling and solution of problems concerning optimal placement of facilities in order to minimize transportation costs, outperform competitors facilities, etc. Our focus area in Facility Location Problems is the *minsum facility location*. In a basic formulation, the Facility Location problem consists of a set of potential facility sites L where a facility can be opened, and a set of demand points $D = \{x_1, x_2, ..., x_d\}$ that must be serviced. The goal is to pick a subset $F = \{f_1, ..., f_n\}$ of facilities to open, to minimize the sum of distances from each demand point to its nearest facility, plus the sum of zeta (z), the opening costs of the facilities.

$$\underset{D}{\operatorname{argmin}} \quad z|F| + \sum_{i=0}^{n} \sum_{x \in D} d(x, f_i)$$

As we mentioned in Section 2.6 the algorithm we have implemented is actually a Local Search solving the *Facility Location Problem*, as a subroutine, for better performance. A new task in this local search algorithm is the binary search of the facility cost (*zeta*), in order to have the desirable number of clusters at the end of the process.

We have implemented the facility location algorithm in two MapReduce stages.

4.2 Description of the Algorithm

Firstly, we implemented a stream clustering algorithm based on hierarchical scheme, discussed earlier in Section 2.6.1. According to this scheme the intermediate-level clustering is made by a local search algorithm solving the facility location problem with a binary search of facility cost zeta. Our implementation focuses

on this clustering of the intermediate-level and especially on the Facility Location algorithm, we proposed a parallel version for Hadoop Online Prototype.

As discussed earlier, the Local Search algorithm that Guha et al. propose in their paper [8] begins finding an initial solution, a list of feasible points, and a range upon the binary search for facility cost runs, before the *Facility Location* subroutine can run. So, in our implementation we begin with computing some important parameters that will use later. First of all we compute a zeta factor that is later used as the initial maximum facility opening cost in a range needed by binary search. Guha et al. propose this zeta initialization to be calculated by simply adding all point distances from one random chosen point in our data set. Then, we need a good initial solution to start our algorithm, which is also proposed to be computed with a simple algorithm shown in the pseudocode below. We create a cluster center at the first point and then for every point after the first, depending on its nearest distance (d_{min}) from the existing cluster centers, we create, with a probability $d_{min}/zeta$, a new cluster center at this point. Otherwise, we add this current point to the nearest existing cluster.

Algorithm 4.2.1: INITIALSOLUTION($N, zeta$)

local *clusterList*
local *currentPoint*
local *nearestDist*, *nearestClu*
local *probability*

clusterList ← ADD(*N.readFirst*)

while HASNEXT(N)
do
- *currentPoint* ← *N.next*
- *nearestDist* ← FINDNEARESTCLU(*clusterList*)
- *propability* ← *nearestDist*/*zeta*
- **if** *probability* ← **true**
  - **then** {*clusterList* ← ADD(*currentPoint*)
  - **else** {*nearestCLu* ← ASSIGNPOINT(*currentPoint*)

Initial Clusters and *Facility Cost* (*zeta*) are two of the parameters we need to run the *Facility Location* algorithm. A third parameter is a list of feasible centers. This list is a randomly chosen set of $\Theta(\frac{1}{p}\log k)$ points and it is used to evaluate gain, the cost we would save or add in case a new point becomes a facility, instead of evaluating it on every point of our data set, as Guha et al. propose in their paper [8] in order to reduce the total running time of their algorithm. By that way it is still likely to choose good medians but will finish our computation sooner. On this set of feasible centers, the p value is depending on the smaller cluster in our data set. More specifically, $|C_i|/|N| \geq p$, where C_i is the set of points in N assigned to c_i assuming that the points $c_1, ..., c_k$

constitute an optimal solution to the k-Median problem for the data set N. However, we have computed our initial solution and through this cluster centers we take the smaller cluster size.

After we have computed the facility cost and have drawn some initial clusters and a list of feasible centers from our data set, we run iteratively the facility location subroutine and a binary search, after the *FL*, for the better choice of zeta (facility cost) in case the desired number of clusters is not achieved. This iteration step works until our cluster centers are exactly k (a number of our choice - it is an input parameter) or the minimum zeta becomes greater than a degree (also our choice - input parameter) of the maximum zeta in binary search, as it begins searching in a range of zero bottom and the initial zeta as top. This iteration also breaks when our clusters centers are more than k and less than *2k*.

The above logic is shown in the pseudocode below.

Algorithm LSEARCH(N**,** $d(.,.)$**,** k**,** ϵ**,** ϵ'**,** ϵ'' **)**

1. $z_{min} \leftarrow 0$.
2. $z_{max} \leftarrow \sum_{x \in N} d(x, x_0)$ (for x_0) an arbitrary point in N).
3. $z \leftarrow (z_{min} + z_{max})/2$.
4. $(I, \alpha) \leftarrow$ InitialSolution(N, z)
5. Randomly pick $\Theta(\frac{1}{p} \log k)$ points as feasible medians.
6. While medians $\neq k$ and $z_{min} < (1 - \epsilon'')z_{max}$:
   - Let (F, g) be the current solution.
   - Run $FL(N, d, \epsilon, (F, g))$ to obtain a new solution (F', g')
   - If $k \leq |F'| \leq 2k$, then exit loop.
   - If $|F'| > 2k$, then $z_{min} \leftarrow z$ and $z \leftarrow (z_{max} + z_{min})/2$;
   - else if $|F'| < k$, then $z_{max} \leftarrow z$ and $z \leftarrow (z_{max} + z_{min})/2$;
7. Return our Solution (F', g').

Our MapReduce implementation focuses on the Facility Location sub algorithm. The *Facility Location* clustering implementation is divided in two MapReduce stages. The first stage runs for a number (given parameter) of feasible points in a feasible list and calculates 4 statistics that we later use, outside Hadoop, to compute the gain or loss we would have if the current feasible point becomes a cluster center. We find, for all points, the nearest cluster center and, in comparison to the Euclidean distance with the feasible point returns four sums: the first sum is the total distance from all points to the current existing nearest cluster center, the second sum is the total distance from all points to the current feasible point, the third is the total distance from points, closer to the feasible, to the nearest existing cluster center and the last is the total distance from points, closer to the feasible, to the current feasible.

Figure 4.1 depicts the two last totals. The total of gray lines is the third sum, while the total of dashed lines is the fourth sum.

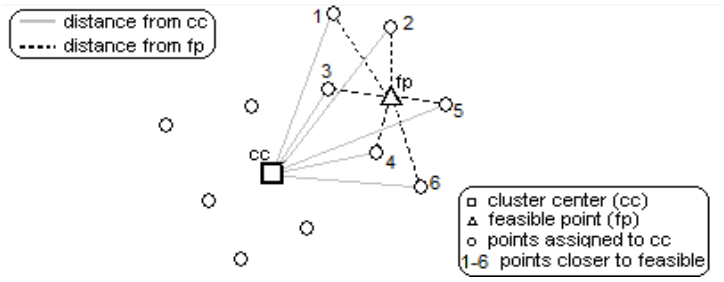

Figure 4.1: Distances from points closer to feasible, to center and to feasible.

Gain for each candidate feasible point is computed in a function outside Hadoop, using a function that is called once for each cluster, as the reducer emits the statistics for each cluster. This function works as shown in the pseudocode (Gain-Function) below. Once gain-function is completed we have a positive gain if the existence of current feasible is more useful to distribute our input data points (or, a zero gain otherwise).

Symbol	Definition
AC	Sum of **A**ll distances from assigned points to **C**enter of cluster
AF	Sum of **A**ll distances from assigned points to **F**easible point
SC	Sum of (**S**ome) distances from points* to **C**enter of cluster
SF	Sum of (**S**ome) distances from points* to **F**easible point
SC'	Sum of (**S**ome) distances from points** to **C**enter of cluster
SF'	Sum of (**S**ome) distances from points** to **F**easible point
	*closer to the feasible than the center of cluster
	**closer to the center of cluster than the feasible

Table 4.1: Definition of variables.

Algorithm 4.2.2: GAIN-FUNCTION($AC, AF, SC, SF, cluster, zeta, fnum$)

```
local cost
local totSF

totSF ← totSF + SF
if SF != 0   // At least one point closer to feasible
  then { cost ← cost + SF − SC
         if SF′ − SC′ < zeta   // Check not closer to feasible points
           then { cost ← cost + SF′ − SC′ − zeta
                  DELETE(cluster)
  else { if AC == 0    // only one point - center itself
           then { if AF < zeta
                    then { cost ← cost + AF − zeta
                           DELETE(cluster)
if totSF == 0// None is closer to feasible
  then { if distance(nearestClu.center,feasible)*fnum> zeta
           then cost ← cost−distance(nearestClu.center,feasible)*fnum
```

Note that none of our data points are actually reassigned yet. Reassignment is our next step in second MapReduce Stage. This stage is called only when gain is positive, otherwise there is no reason to reassign points to the same previous cluster centers, as feasible point will not be included in cluster list and nothing will change. So, this second stage finds the minimum distance of all points to the cluster centers and the feasible and assigns it to the corresponding cluster. In case the minimum distance is to the feasible a new cluster is created and later added to the list of clusters.

The driver routine that calls the two Map/Reduce Jobs is shown in the Figure 4.2.

In the remainder of this chapter, we give the detailed description of a custom Writable variable, *Cluster*, that we used and the extended description of each one of the two stages of the algorithm along with their pseudocode. In every map/reduce stage described below, we give the types of input and output variables, and also an example of the key/value pairs that are read and written. The notation that is used in the following sections to express the key/value pairs of the map/reduce framework uses underlining for keys and parenthesis for values, i.e., key, (value).

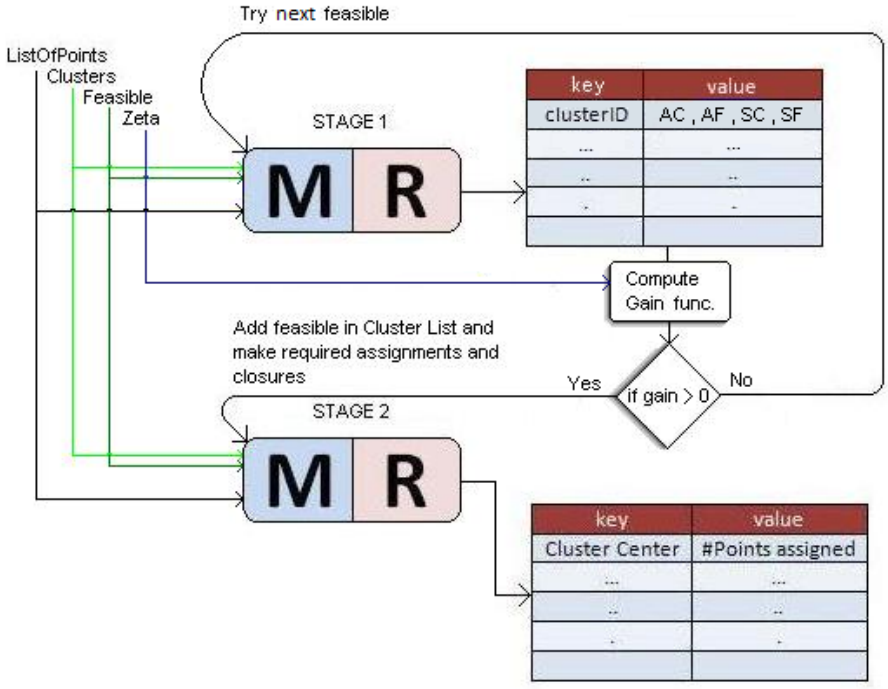

Figure 4.2: Driver routine for Facility Location algorithm.

4.2.1 Custom Writable Variable

This Local Search algorithm is also dealing with clustering issues, so the need arises for a composite variable that keeps all the appropriate information for a cluster and the points assigned to it. This *Cluster Writable* variable differs a little with the one we used in our earlier K-Means implementation.

Cluster Writable

Also in this *Facility Location* algorithm, our need to categorize clusters based on their center and their assigned points led us to implement a custom Writable variable, which we called "*Cluster*". This variable is created in order to help us emit composite values and is used to express clusters. It contains five fields: clusterId, center, numPoints, pointTotal, sqrTotal. clusterId is the unique identity of a cluster, center is a vector of doubles expressing the dimensions of the cluster center, numPoints is an integer expressing the number of points this cluster is consisted of, pointTotal is a vector of doubles expressing the sum per dimension of all points assigned in cluster and sqrTotal is a vector of doubles expressing the sum per squared dimension of all points assigned to cluster center, those two lasts vectors is used in the end to show the efficiency of clustering. In

this algorithm we do not recompute the new cluster centers upon the previous centers and also there is no convergence factor so we do not need the converged variable that we had in K-Means.

Supposing that we have two points, let us say *p1*:{1,1} and *p2*:{2,1} assigned to the same *cluster center*:{2,2} then the *Cluster* attributes would be: clusterId = 1 (random choice but different from other existing objects of *Cluster*), center = {2,2}, numPoints = 2, pointTotal = {3 ($p1_x$+$p2_x$) , 2 ($p1_y$+$p2_y$)}, sqrDistance = 5 ($p1_x^2 + p2_x^2$) , 2 ($p1_y^2 + p2_y^2$).

Input Data		Cluster				
point		*id*	*center*	*num*	*pntTot*	*sqrTot*
1	1	1	[2 , 2]	2	[3 , 2]	[5 , 2]
2	1					

The class *Cluster* implements the following methods:

- associated constructors: All Writable implementations must have a default constructor so that the MapReduce framework can instatiate them.
- setters and getters for the variables.
- *toString*(): returns the String representation of the variable. It is called to write the reducer's output to the HDFS filesystem, if the output format is TextOutputFormat.
- *readFields*(): deserializes the bytes from the input stream by delegating to each object. It is called by a mapper to read from the HDFS.
- *write*(): serializes each object in turn to the output stream. It is called by the collect() function and writes the variable to the HDFS, if the output format is SequenceFileOutputFormat.

4.2.2 Facility Location In Map/Reduce

As we discussed before Facility Location algorithm consists of two Map/Reduce stages. The first stage of our algorithm is executed iteratively until a desired number of feasible points are checked. In the intermediate of these iterations, a function is executed checking whether the addition of the feasible point, (processed in the last iteration) in the cluster list would be gainful for the distribution of our data set or not. If the feasible helps to reduce the total cost, then the second stage of our algorithm is activated assigning all points to its nearest cluster, including the one that feasible has just created.

The algorithm takes as input an initial Solution, a facility cost (zeta), and a list of feasible points.

FL Stage 1

The first stage of our implementation calculates some statistics, for the clusters, that we later use, in an outside Hadoop function, to compute gain for each feasible center. The first iteration, where this stage is called, begins after the computation of an Initial Solution and a random choice of a standard number of feasible centers. So this stage, apart from the data set, has as inputs a list of clusters, and a list of feasibles.

Map

Mapper interface has four parameters that specify the input and output key/value pairs. In this Map function the input key is a long integer offset, and the input value is a line of text containing one point. This mapper's mission is to evaluate two statistics, with the knowledge of the current cluster list (a global liist) and the feasible point: the first statistic is the distance from the input point to the nearest existing cluster center and the second is the distance to the feasible point. Those two distances concatenated to one text type variable are emitted as the output value. The output key is also a text type variable expressing the id of the nearest cluster found. This map function actually assigns each point to a cluster but also keeps the distance to the feasible, so we can later be able to compare the sum of distances per cluster.

InputTypes : LongWritable , (Text)

OutputTypes : Text , (Text)

Our algorithm is designed for data points that are all vectors with the same dimension. As discussed in section 2.6 our implementation works on a hierarchical scheme, so at the process of first-level datasets, mappers read data of type "*x y m1*", because all points have multiplicity "1" (one). However, at the intermediate levels where data have been computed from previous jobs, mappers work on data of type "*x y mn mtotx toty mtotx*^{2} *toty*^{2}". This data type includes further information about those points that are previously assigned to this, hidden-in-data, point. Examples of both input types are shown after this mapper's analysis. So, once a mapper reads a line splits to its components. One of these components is the data point that mapper uses to find the distances from nearest cluster center and from the feasible point. Once distances have been computed, mapper emits the id of the nearest cluster (as output key) and a Text type object containing the point's nearest distance from all cluster centers and the distance from feasible point (as a value). Moreover, before emitting, distances are multiplied according to the multiplicity factor of the data point. More specifically, the output value is of a type *A,B*, where A is the nearest distance of data point to all cluster centers and B is the distance between data point and feasible.

The pseudocode below depicts the logic of our first-stage mapper.

Algorithm 4.2.3: FLMapperStg1($key, value$)

global $listOfClusters$
global $feasible$
local $point, multi$
local $nearestDistance, feasibleDistance$
local $nearestCenter$
local $keyOutput, valueOutput$
local A, B

$point, multi \leftarrow$ SPLIT($value$)

$feasibleDistance \leftarrow$ DISTANCE($feasible, point$)

for each $cluster \in listOfClusters$
do {
  $center \leftarrow$ GETCENTER($cluster$)
  $distance \leftarrow$ DISTANCE($center, point$)
  if $distance < nearestDistance$
    then {
      $nearestDistance \leftarrow distance$
      $nearestCluster \leftarrow cluster$
    }
}

$A \leftarrow nearestDistance * multi$
$B \leftarrow feasibleDistance * multi$

$keyOutput \leftarrow key$

$valueOutput \leftarrow A, B$

output ($keyOutput, valueOutput$)

Example : If we suppose that we have from initialization the cluster centers: [1 , 1],[1 , 6], with ids 1, 2 respectively, and a set of first-level data points in input file, as it is shown below, and the feasible: [5 , 3] then the outcome of the map execution would be this:

input				*id*	*A*		*B*
1	5	$m2$	$\rightarrow$	$\underline{2}$,	(2	,	8.94)
1	1	$m1$	$\rightarrow$	$\underline{1}$,	(0	,	4.47)
1	2	$m1$	$\rightarrow$	$\underline{1}$,	(1	,	4.12)
1	6	$m1$	$\rightarrow$	$\underline{2}$,	(0	,	5)
5	3	$m1$	$\rightarrow$	$\underline{1}$,	(4.47	,	0)

$$5 \;\; 4 \;\; m3 \;\; \rightarrow \;\; \underline{2} \;, \; (\; 13.41 \;\; , \;\; 3 \;\;\;)$$

Now, if we have n-level data points the result will be the same, as the additional information these data have is not used in this step of our algorithm.

$$1 \;\; 5 \;\; m2 \; m3 \; 9 \; m5 \; 41 \;\; \rightarrow \;\; \underline{2} \;, \; (\;\; 2 \;\;\; , \;\; 8.94 \;\;)$$

$$1 \;\; 1 \;\; m1 \; m3 \; 3 \; m5 \; 5 \;\; \rightarrow \;\; \underline{1} \;, \; (\;\; 0 \;\;\; , \;\; 4.47 \;\;)$$

Reduce

As we said in the beginning of this section this stage is used for statistical information. This reducer takes as input key/value pairs two text-type variables, as the above mapper's output and returns as output key the same id of the cluster it is processing, and as output value a text-type variable, in which four statistics about the current cluster are concatenated. This reducer is actually a very simple one, as it sums in a way the values in the value list.

InputTypes : Text , (Text)

OutputTypes : Text , (Text)

Each reducer in this stage reads as input, data of type {**key** , [($nearestClu1$, $distFeas1$),($nearestClu2$, $distFeas2$),...,($nearestCluN$, $distFeasN$)]}. Reducers for each key, actually creates four totals. The one sums all $nearestClu$ values, expressing the sum of distances to the center of all points assigned to this cluster, the second one, sums all $distFeas$ values, expressing the sum of distances to the feasible point of all points. The third, sums those $nearestClu$ values that in comparison with the corresponding value $distFeas$ are greater. The fourth and last total, keeps the sum of those $distFeas$ distances that in comparison with the corresponding $nearestClu$ are smaller. Practically, the first total keeps the cost of cluster, in case feasible would not exist, the second keeps the cost of cluster in the case of the feasible to be the center, the third keeps the cost of cluster, in case feasible exists as a cluster and has taken some points from it, and the last keeps the cost of cluster that feasible would create and would take some the current cluster's points to itself.

The algorithm below shows how this reducer works.

Algorithm 4.2.4: FLReducerStg1$(key, value)$

local $nearestAll, feasibleAll$
local $nearestSome, feasibleSome$
local $prevDist, feasDist$
local $outputValue$

$nearestAll \leftarrow 0$
$feasibleAll \leftarrow 0$
$nearestSome \leftarrow 0$
$feasibleSome \leftarrow 0$

while hasNext$(valueList)$
do
- $prevDist, feasDist \leftarrow$ split$(value)$
- $nearestAll \leftarrow$ add$(prevDist)$
- $feasibleAll \leftarrow$ add$(feasDist)$
- **if** $feasDist < prevDist$
  - **then**
    - $nearestSome \leftarrow$ add$(prevDist)$
    - $feasibleSome \leftarrow$ add$(feasDist)$

$outputValue \leftarrow nearestAll, feasibleAll, nearestSome, feasibleSome$

output $(key, outputValue)$

Example : This reduce method takes as input the output of the above map method. So if we suppose the same data set as input and the output of the map method, then the outcome of the reduce step would be this:

key	*valueList*	*nAll*	*fAll*	*nSome*	*fSome*
1	(0 , 4.47)	1	5.57	8.59	4.47	0
1	(1 , 4.12)					
1	(4.47 , 0)					
2	(2 , 8.94)	2	15.41	16.94	13.41	3
2	(0 , 5)					
2	(13.41, 3)					

4.2.3 FL Stage 2

This second MapReduce Stage of our Facility Location is called once for each feasible point, if only the gain in cost is positive. In this stage we reassign our

data points to the updated list of cluster centers (including the feasible point). We find for each point the minimum distance to the centers and assign it to the correct cluster.

Map

This mapper takes as input value a text-type variable, which actually is a line from our data set and contains a single point. The input key is a long integer offset. This mapper's mission is to find for each point the smaller distance from either one of the centers from in the list of clusters or the feasible point. All mappers know the current cluster list and the fesaible point. In the end the output key is text-type parameter expressing the cluster or the feasible that is closer to the processing point, and the output value is also a text-type parameter expressing the point with some useful informations that we later use in the reduce step.

InputTypes : LongWritable , (Text)

OutputTypes : Text , (Text)

The input of this map method is a file containing a set of data points. According to the hierarchical scheme, our algorithm works on as mentioned in section 2.6, the data set that reducers receives as input are of two kinds; first-level and intermediate-level type. At the process of the first-level set of data points, mappers read data of type "*x y m1*", because all points have multiplicity "1" (one) . At the intermediate-levels, mappers read data of type "$x\ y\ mn\ mtotx\ toty\ mtotx^2\ toty^2$". If we split this data on m, we can see that this type of intermediate data includes, apart from the data point, its multiplicity, which actually is the number of points that assigned to this point (as cluster center) in a previous level, and two other vectors being more information about the cluster and its assigned points from the previous step. Once the mapper reads the data point from the splitted input, it computes the distance (from this point) to the known feasible and searches in a global known cluster list the nearest cluster. Once, those two distances have been computed, mapper compares them and emits as output key the id of nearest cluster or an "f" (for feasible), depending which one is closer to the data point. The output value is the input data point with its all information. In order to take advantage of the number of our working machines and not to face a bottleneck at this point with a limited number of reducers, we tweak the output key concatanating the id with a random number between 1 and the number of working machines divided by the size of cluster list. So, we distribute the intermediate data to much more reducers than before, when only a number of cluster list's size reducers would have to bear the hole process letting the rest remain idle.

The pseudocode below shows the way a mapper works.

Algorithm 4.2.5: FLMapperStg2($key, value$)

global $listOfClusters$
global $feasible$
global $reducers$
local $point$
local $nearestDistance, feasibleDistance$
local $nearestCenter$
local $keyOutput, valueOutput$
local id

$point \leftarrow$ SPLIT($value$)

$feasibleDistance \leftarrow$ DISTANCE($feasible, point$)

for each $cluster \in listOfClusters$
do {
  $center \leftarrow$ GETCENTER($cluster$)
  $distance \leftarrow$ DISTANCE($center, point$)
  if $distance < nearestDistance$
    then { $nearestDistance \leftarrow distance$
               $nearestCluster \leftarrow cluster$

if $feasibleDistance < nearestDistance$
  then { $id \leftarrow f$
  else { $id \leftarrow$ GETID($nearestCluster$)

$random \leftarrow$ RAND($reducers/(listOfClusters.size() + 1)$)
$keyOutput \leftarrow id + random$
$valueOutput \leftarrow value$

output ($keyOutput, valueOutput$)

Example : If we suppose that we have from initialization the cluster centers: [1 , 1][1 , 6], with ids 1, 2 respectively, a set of data points in input file and as feasible: [5 , 3] then the outcome of the map execution would be this:

$$
\begin{array}{llll}
1\ 5\ m2 & \rightarrow & \underline{2r}\ , & (\ 1\ 5\ 2\ ) \\
1\ 1\ m1 & \rightarrow & \underline{1r}\ , & (\ 1\ 1\ 1\ ) \\
1\ 2\ m1 & \rightarrow & \underline{1r}\ , & (\ 1\ 2\ 1\ ) \\
1\ 6\ m1 & \rightarrow & \underline{2r}\ , & (\ 1\ 6\ 1\ ) \\
5\ 3\ m1 & \rightarrow & \underline{fr}\ , & (\ 5\ 3\ 1\ ) \\
5\ 4\ m3 & \rightarrow & \underline{fr}\ , & (\ 5\ 4\ 3\ )
\end{array}
$$

We can see that output keys are 1r , 2r in case one of the existing clusters is closer to the point and "fr" when the distance of the point to feasible is shorter than to any other existing cluster centers. The "r" character, as we said before, is a random number that helps us distribute our data to more reducers. In this case r can take values between 1 and $numberOfReducers/(2 + 1)$. For example, if our job runs 90 reducers then each cluster's data will be distributed to 30 reducers instead of 1. Also, in the output values the character m is missing as in the beginning of the mapper we split the input and then we create a new variable without it, which is the output value.

Reduce

This reducer is the final step of all points assignment to the clusters including the new cluster that feasible created. Reducer takes as input key and values two text-type variables, the same as the mapper's output key/value pairs. The output key is also a text-type variable expressing the id of the cluster that we are processing and the output value is Cluster-type variable (custom Writable variable, we implemented), that keeps all the information needed about the cluster.

InputTypes : Text , (Text)

OutputTypes : Text , (Text)

Reducers take as input the output produced by the mappers. After, the output data of mappers, has been shuffled and sorted, it has been grouped by key. Each reducer takes a unique key with all the values that share the same key. This means that each time reducer handles the data of a whole cluster. In this reducer the key is a cluster id or an "f", which means that it processes the data of the new cluster that feasible will create. Once reducer receives the data, the input key is searched in the current cluster list to match the appropriate cluster. However, the cluster list is not updated yet, as the feasible has not create a cluster yet and no reassignments have been done. So the search in cluster list will not return a value if the key is "f". In case the data belongs to the feasible, reducers need to create a cluster first. Otherwise reducers continue their job as normally, run on the value list and assign all points to the cluster found. Finally, the output key is the cluster id and the output value is the cluster itself.

The pseudocode below show how this reducer works.

Algorithm 4.2.6: FLReducerStg2($key, value$)

global $listOfClusters$
local $valueIn$
local $outptuKey, outputValue$

while hasNext($valueList$)
do $\begin{cases} valueIn \leftarrow \text{getNext}(valueList) \\ \text{addPoints}(cluster, valueIn) \end{cases}$

$outputKey \leftarrow \text{getId}(key)$
$outputValue \leftarrow cluster$

output ($outptuKey, outputValue$)

Example : This reduce method takes as input the output of the above map method. So if we suppose the same data set as input and the output of the map method, then the outcome of the reduce step would be this:

$$\left.\begin{array}{l} \underline{1}\ ,\ (\ 1\ \ 1\ \ 1\ ) \\ \underline{1}\ ,\ (\ 1\ \ 2\ \ 1\ ) \end{array}\right\} \quad \underline{1}\ ,\ (\ 1\ [\ 1\ \ 1\ ]\ 2\ [\ 2\ \ 3\ ]\ [\ 5\ \ 2\ ]\ )$$

$$\left.\begin{array}{l} \underline{2}\ ,\ (\ 1\ \ 6\ \ 1\ ) \\ \underline{2}\ ,\ (\ 1\ \ 5\ \ 2\ ) \end{array}\right\} \quad \underline{2}\ ,\ (\ 2\ [\ 1\ 6\ ]\ 3\ [\ 2\ 11\ ]\ [\ 2\ 61\ ]\ )$$

$$\left.\begin{array}{l} \underline{f}\ ,\ (\ 5\ \ 3\ \ 1\ ) \\ \underline{f}\ ,\ (\ 5\ \ 4\ \ 3\ ) \end{array}\right\} \quad \underline{f}\ ,\ (-1\ [\ 5\ 3\ ]\ 4\ [\ 10\ 7\ ]\ [\ 50\ 21\ ]\ )$$

In the example above, we see the clusters above on the right side, with the only strange the id of the newly created cluster with the feasible point as center. In this case we have to initialize the id with an integer as we can not leave the f for obvious reasons, this choice, at the moment, is a negative one as we do not know how far the other ids have been and we do not want any duplicates, as ids are only positive. As we said before, this negative id is temporary. The final id of the new cluster will be given outside Hadoop, where we keep a continuous rising variable for that reason.

Also, in the example above we have not included the concatenated random variable to the input key for simplicity purpose. Outside Hadoop there is a loop that sums all the reducer's outputs for each cluster (due to this random variable we would have more than one output) creating the final clusters.

Note that no deletion is made in this Map/Reduce step. The algorithm says that points cannot be reassigned except to the feasible point, so we keep all clusters at this step and with an other function outside Hadoop we check if the remaining assigned points to each cluster are costly enough to handle the facility cost. Otherwise we reassign all the points to the feasible first and then delete the cluster.

Chapter 5

Results

In this chapter we present and analyze the results of our experiments. We conducted all experiments on a 13-machine cluster (1 master - 12 slaves) with Quad Core Processors and 4GB RAM. In our experiments we compare the SSQ measure, which is the sum of *squares* of distances to medians, and the scalability of our algorithms to the Hadoop framework. Our experiments are divided into 2 parts. The first part checks the sensitivity of some basic parameters, that we use in our algorithms and how strong or weak influence they have on the final results as far the quality of clustering (SSQ) and time of completion. The second checks the scalability of our algorithms on time either if we change the size of datasets or the number of running nodes on cluster.

The dataset that we used contains real data found on a Data Mining Community's Resource [3] on Internet.

Our dataset contains statistics for words that have been found in books. Each line in this dataset consists of four variables: one expressing the year that the statistics for each word is generated, the second variable is the number of overall times occurred this word, the third expresses the distinct number of pages this word is found and last is the number of distinct books this word has been written to. So we have in each line a four-dimension point.

These datasets were generated in July 2009 by Google labs.

5.1 Sensitivity

In this section we experiment on some basic parameters, we use in our algorithms, to check how much they infect the final outcome as far the quality of clustering and the total time of completion, so to run, in the next step, some time-performance experiments with the best selection of parameters.

K-Means algorithm runs on two basic parameters: maximum number of iterations and a factor delta that is used for checking the convergence of the previous to new cluster centers in each iteration. Firstly, we fixed the number of iterations and run our K-Means algorithm for different delta factors. We

experimented on three different datasets to see also the impact of delta on different composition of data. All experiments have been repeated 4 times because our algorithms begin on random cluster centers and we want to check how this randomization affect the final outcome, and also have a better view of our results.

For the quality of clustering we focus on SSQ and how many clusters have been converged. In the tables below we show the number of converged clusters with a variable that is *F, where * is the number of not converged clusters or simple F when none of the clusters have been converged and T when all clusters are converged.

Fixed Iterations

Running on our hadoop cluster of 13 nodes and examining the delta factor with fixed maximum iterations to be 15, we get the following results.

Data	**SSQ**	**converge**	**Time (msec)**
1 MB	3.00E+007	T	223854
45000 points	3.00E+007	2F	298294
3-dimension	3.00E+007	2F	306957
	3.00E+007	2F	306498
Average	**3.00E+007**	**2F**	**284015 ms**

Table 5.1: Few dimensions with delta 0.2

In Table above (5.1) we see the results from 4 same experiments run on 1MB data containing 45000 3-dimensional points. The algorithm runs with delta factor 0.2, and fixed maximum iteration to be 15.

Data	**SSQ**	**converge**	**Time (msec)**
1 MB	4.02E+007	F	314302
13800 points	3.98E+007	2F	314534
9-dimension	4.00E+007	2F	311555
	3.99E+007	2F	311780
Average	**4.00E+007**	**2F**	**313042 ms**

Table 5.2: Many dimensions with delta 0.2

Data	SSQ	converge	Time (msec)
3.3 MB	1.37E+008	1F	344166
45000 points	1.39E+008	F	336537
9-dimension	1.38E+008	F	327090
	1.38E+008	F	338441
Average	**1.38E+008**	**F**	**336558 ms**

Table 5.3: Many dimensions with delta 0.2

In two Tables above (5.2, 5.3) we run on 9-dimensional points. In the first Table we experiment on a *same-sized* data with the 3-dimension dataset (1MB 9-dim – 1MB 3-dim) in the beginning. While in the second, we run on a *same-amount* of points with the 3-dimension dataset (45000 9-dim points – 45000 3-dim points). We can see that this delta (0.2) demands a strong-accuracy for our final clusters, but 15 iterations are not enough to allow the algorithm converge on this result.

In the next three (5.4, 5.5, 5.6) Tables we run on the same datasets changing the delta to 0.5. We want to check what will happen if we sacrifice a kind of accuracy.

Data	SSQ	converge	Time (msec)
1 MB	3.00E+007	1F	308352
45000 points	3.00E+007	T	71695
3-dimension	3.00E+007	T	268918
	3.01E+007	2F	309252
Average	**3.00E+007**	**1F**	**239554 ms**

Table 5.4: Few dimensions with delta 0.5

Data	SSQ	converge	Time (msec)
1 MB	3.97E+007	T	290176
13800 points	3.98E+007	2F	319935
9-dimension	3.99E+007	2F	323985
	3.98E+007	1F	316524
Average	**3.98E+007**	**1F**	**312655 ms**

Table 5.5: Many dimensions with delta 0.5

Data	SSQ	converge	Time (msec)
3.3 MB	1.37E+008	1F	332744
45000 points	1.37E+008	F	335331
9-dimension	1.39E+008	2F	354608
	1.37E+008	F	308352
Average	**1.37E+008**	**2F**	**332758 ms**

Table 5.6: Many dimensions with delta 0.5

In this case we can see that our datasets converge easier than before and the average time of completion is similar or better. We do not lose in accuracy as in this number of iterations the previous delta (0.5) almost never converged so the computation has no difference. The difference in accuracy would be appeared if we had the ability to let our algorithm run until it converges without limit of iterations, then the final clusters would be better as the accuracy factor (delta) would be stronger. Now with a greater delta, we have the advantage of completion time and convergence.

In the next three Tables (5.7, 5.8, 5.9) we try an even greater delta 0.8 to see if the results are also encouraging.

Data	SSQ	converge	Time (msec)
1 MB	3.02E+007	2F	308790
45000 points	3.00E+007	T	277997
3-dimension	3.01E+007	T	252633
	3.01E+007	T	273928
Average	**3.00E+007**	**T**	**278337 ms**

Table 5.7: Few dimensions with delta 0.8

Data	SSQ	converge	Time (msec)
1 MB	3.98E+007	T	274211
13800 points	4.03E+007	1F	316817
9-dimension	4.01E+007	T	194161
	4.00E+007	1F	318165
Average	**4.01E+007**	**T**	**275838 ms**

Table 5.8: Many dimensions with delta 0.8

Data	SSQ	converge	Time (msec)
3.3 MB	1.37E+008	T	309568
45000 points	1.39E+008	T	184483
9-dimension	1.37E+008	2F	332950
	1.40E+008	2F	358312
Average	**1.38E+008**	**1F**	**306328 ms**

Table 5.9: Many dimensions with delta 0.8

From the experiments above we can see that our algorithm converges quite easier and faster with delta=0.8. However, some times we see that the process is completed quite fast because of convergence but the SSQ is higher than we expected i.e., in Table 5.8 the algorithm has converged in 194 *sec*, with *SSQ* 4.01E+007, in contrary to Table 5.5 that it converged in almost 290 *sec* but the *SSQ* is a lot better. The disadvantage of a greater delta is that it probably ends the process earlier with not the desirable quality in results.

Fixed Delta

So after the previous experiments, we fix the delta factor to be 0.5 and experiment on different maximum number of iterations. We want to see how this parameter (max. iterations) extends or decreases the processing time and how important this change is to the quality of clustering. We also use three different datasets as in the previous experiments.

Data	SSQ	converge	Time (msec)
1 MB	3.01E+007	2F	222076
45000 points	3.00E+007	2F	222749
3-dimension	3.00E+007	1F	222631
	3.00E+007	T	94795
Average	**3.00E+007**	**1F**	**190562 ms**

Table 5.10: Few dimensions with 10 maximum iterations

Here we can see that it is very rare to see our clusters converging. Although, the one of the runs has been converged, we see that completion time is very short and this statistic can lead us to the conclusion that we were very lucky with the selection of the initial (random) cluster centers.

In the next two Tables (5.11, 5.12), that we run on 9-dimensional points, we see that it is too difficult for our algorithms to finish due to convergence and definitely the 10 iterations are not enough.

Data	SSQ	converge	Time (msec)
1 MB	4.00E+007	2F	209130
13800 points	3.98E+007	2F	208184
9-dimension	4.06E+007	2F	208189
	3.99E+007	2F	207392
Average	**4.01E+007**	**2F**	**208223 ms**

Table 5.11: Many dimensions with 10 maximum iterations

Data	SSQ	converge	Time (msec)
3.3 MB	1.37E+008	F	226166
45000 points	1.39E+008	2F	230546
9-dimension	1.37E+008	2F	229408
	1.37E+008	2F	245430
Average	**1.38E+008**	**2F**	**232887 ms**

Table 5.12: Many dimensions with 10 maximum iterations

At the next experiments we use 20 iterations as limit for our algorithm, as we saw that 10 are not enough.

Data	SSQ	converge	Time (msec)
1 MB	3.00E+007	T	378696
45000 points	3.00E+007	T	255828
3-dimension	3.00E+007	T	299843
	3.00E+007	T	356421
Average	**3.00E+007**	**T**	**322697 ms**

Table 5.13: Few dimensions with 20 maximum iterations

In Table 5.13 we see that dataset with 3-dimensional points is always completed with all clusters converged. This is very encouraging and allows us to experiment with less iterations for even better results, for such type of datasets.

The 9-dimensional datasets (5.14, 5.15) are not so successful as the first dataset, but their results are also good enough. The average of only one not converged cluster in a multi-dimensional dataset is quite good.

Data	SSQ	converge	Time (msec)
1 MB	3.99E+007	2F	408988
13800 points	3.98E+007	T	305886
9-dimension	3.98E+007	T	404066
	3.98E+007	1F	410966
Average	**3.98E+007**	**1F**	**382476 ms**

Table 5.14: Many dimensions with 20 maximum iterations

Data	SSQ	converge	Time (msec)
3.3 MB	1.37E+008	T	273303
45000 points	1.39E+008	F	445581
9-dimension	1.37E+008	1F	454715
	1.37E+008	T	284359
Average	**1.37E+008**	**1F**	**364489 ms**

Table 5.15: Many dimensions with 20 maximum iterations

As the experiments of our algorithms performance will run with the first type of dataset containing few-dimensional points, we want to experiment once again with 15 iterations to see if this returns a better completion time with satisfactory convergence results.

Data	SSQ	converge	Time (msec)
1 MB	3.00E+007	T	223205
45000 points	3.00E+007	1F	323049
3-dimension	3.01E+007	2F	327114
	3.00E+007	T	141655
Average	**3.00E+007**	**1F**	**253755 ms**

Table 5.16: Few dimensions with 15 maximum iterations

In Table 5.16 we see the results from our K-Means algorithm that runs on 15 maximum iterations and 0.5 delta factor. We observe that not all the runs have totally converged clusters, but there is a high probability with a lot better average time of completion. So the conclusion is that we choose 15 maximum iteration sacrificing on the probability of not converging but having an important advantage on completion time.

In conclusion, we can say that it is harder for multi-dimensional datasets to clustered than datasets with few-dimension points. Also, the size of dataset is an important factor, as the bigger a dataset is the more difficult to be clustered. Of course, very important is the content of a dataset as far the physical position of the points. A dataset with uniformly-spread points would be harder to clustered.

Our second algorithm *Facility Location* runs on one basic input parameter: the number of testing feasible points. This parameter expresses the number of randomly selected feasible points from a list that is created in the beginning of our algorithm. This parameter is important because a big value may lead to a better quality of clustering, as we would have a large variety of points to select but this will delay our algorithm to complete, while a small value will make our algorithm faster but with higher probability of worst results.

Running on our Hadoop-Online cluster of 13 nodes and examining the number of testing feasible points, we get the following results, for 5 feasible points.

Data	**SSQ**	**Time (msec)**
1 MB	3.32E+007	333638
45000 points	4.12E+007	174579
3-dimension	4.24E+007	405464
	4.65E+007	373105
Average	**4.10E+007**	**321696 ms**

Table 5.17: Few dimensions with 5 feasible points

Data	**SSQ**	**Time (msec)**
1 MB	5.43E+007	598675
13800 points	5.82E+007	653535
9-dimension	5.03E+007	612956
	5.35E+007	590741
Average	**5.40E+007**	**613976 ms**

Table 5.18: Many dimensions with 5 feasible points

In Tables above (5.17, 5.18) we see that it is more difficult for our algorithm to cluster a dataset with more dimensions, as the average time of completion for same-sized datasets is doubled in 9-dimension dataset.

Data	**SSQ**	**Time (msec)**
3.3 MB	1.63E+008	818710
45000 points	1.79E+008	955364
9-dimension	2.15E+008	588591
	1.84E+008	882307
Average	**1.86E+008**	**811243 ms**

Table 5.19: Many dimensions with 5 feasible points

The next number of feasible points we tried is 10. Raising the number of feasible points we expect better quality in clustering, but worst completion time.

Data	SSQ	Time (msec)
1 MB	3.76E+007	566746
45000 points	3.32E+007	923639
3-dimension	4.20E+007	639073
	3.35E+007	828596
Average	**3.65E+007**	**739513 ms**

Table 5.20: Few dimensions with 10 feasible points

Already in the first Table 5.20 we see that the average SSQ is a lot better, but the time is over-doubled.

Data	SSQ	Time (msec)
1 MB	4.59E+007	1040234
13800 points	4.91E+007	1198574
9-dimension	4.53E+007	965095
	4.44E+007	1236414
Average	**3.72E+011**	**1110079 ms**

Table 5.21: Many dimensions with 10 feasible points

Data	SSQ	Time (msec)
3.3 MB	2.00E+008	1170225
45000 points	1.41E+008	1827828
9-dimension	1.64E+008	1201169
	1.43E+008	1500718
Average	**1.62E+008**	**1424985 ms**

Table 5.22: Many dimensions with 10 feasible points

In 9-dimension datasets, results are quite similar to the first Table. Quality of clustering is better, however, the average time of completion is raised.

The next try is with 15 feasible points. We expect to see similar improvement to quality and further delay to the time.

Data	SSQ	Time (msec)
1 MB	2.65E+007	1357716
45000 points	3.45E+007	942357
3-dimension	3.69E+007	1344360
	3.83E+007	977491
Average	**3.40E+007**	**1155481 ms**

Table 5.23: Few dimensions with 15 feasible points

The first Table (5.23) is not that encouraging, against to what we expected. We have a linear raise to the completion time, while the SSQ is not decreased also linear.

Data	SSQ	Time (msec)
1 MB	4.07E+007	1500341
13800 points	4.31E+007	1593195
9-dimension	4.68E+007	1553709
	4.24E+007	1490635
Average	**4.32E+007**	**1534470 ms**

Table 5.24: Many dimensions with 15 feasible points

Data	SSQ	Time (msec)
3.3 MB	1.70E+008	1569643
45000 points	1.44E+008	1637204
9-dimension	1.48E+008	1725519
	1.59E+008	1593547
Average	**1.55E+008**	**1631478 ms**

Table 5.25: Many dimensions with 15 feasible points

In this third try we see that our algorithm completes clustering with better quality in results, but this slightly improvement is not satisfactory enough to the further delay of time.

In Conclusion we

Splits

Our algorithms are following a streaming model based on piece-meal approach, so we had to check how the number of pieces affect the final outcome. We tried both algorithms run on the same dataset three times, each time with a different number of splits. We used a dataset of 100 MB size, containing 4-dimensional points.

	SSQ	Time
100 MB - 1*100 MB	5.35E+009	1842932 msec
100 MB - 2*50 MB	5.56E+009	1715841 msec
100 MB - 4*25 MB	5.68E+009	1492355 msec

Table 5.26: K-Means , splits

In Table 5.26 we can see that the more splits we have the less time algorithm needs to be completed. We expected that quality will be worsen with the raise of the number of splits. On the other hand, time reduces because, as we saw before in delta and number of iteration experiments, it is easier for a smaller dataset to converge and complete its process before reaching the maximum number of iterations.

	SSQ	Time
100 MB - 1*100 MB	5.11E+009	2678537 msec
100 MB - 2*50 MB	6.88E+009	4113837 msec
100 MB - 4*25 MB	8.59E+009	5413499 msec

Table 5.27: Facility Location , splits

In Table 5.27 neither time, nor the quality is getting better by raising the number of splits. Both quality and time are expected to be worsen. As we mentioned before, Facility Location algorithm, faces a bottleneck in its first stage (gain computation) as the number of reducers is limited. FL runs a great amount of jobs before get completed and most of them are computing gain, so scalability of this algorithm has still room of improvement.

5.2 Performance

Next, we will measure the scale up that we have achieved in our implementations. These experiments are separated into two types. In first type we experimenting on different number of working nodes on the same dataset and checking the total time of completion. The second type also checks the total time of completion but now on different sized dataset with all nodes enabled and working.

Nodes Vs Time

We run our K-Means implementation with a 4-dimension dataset as input, examining clustering with delta = 0.5 and maximum iterations = 15, while changing the number of compute nodes.

The figure below (5.1) summarizes the results above:

Figure 5.1 shows the scale up that we achieves with our algorithm, by running experiments in 4, 8 and 12 compute nodes. As we can see the time is decreasing

	4 nodes	**8 nodes**	**12 nodes**
1GB - 10*100MB	22979223 msec	10910371 msec	7645795 msec
$\sim$ hours	*6.4*	*3*	*2.1*

Table 5.28: K-Means , delta = 0.5 , max.iterations = 15

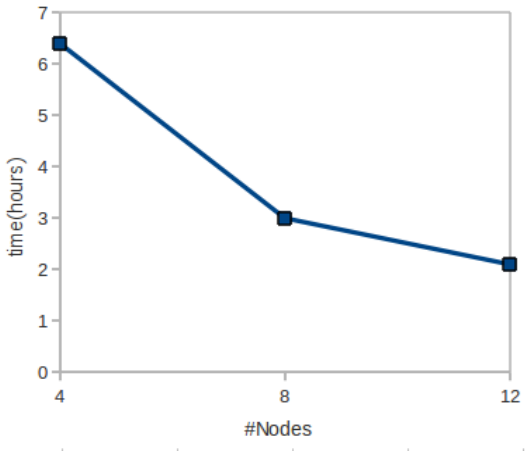

Figure 5.1: Number of compute Nodes Vs Time (K-Means).

linearly, while adding more working nodes.

We repeat the same experiment for our second algorithm, Facility Location, with input a 4-dimension dataset and number of testing feasible point = 10. Tests are occurred on 4, 8 and 12 nodes and results are shown in the Table below.

	4 nodes	**8 nodes**	**12 nodes**
1GB - 10*100MB	88920436 msec	55819384 msec	47461542 msec
$\sim$ hours	*24.7*	*15.5*	*13*

Table 5.29: Facility Location , feasible points = 10

Figures 5.1 and 5.2 show the scale up, or in other words the rank of parallelization that we have achieved, in our implementations. In our second algorithm, time of completion is quite long and the reason is that more than 60 jobs are completed for each run.

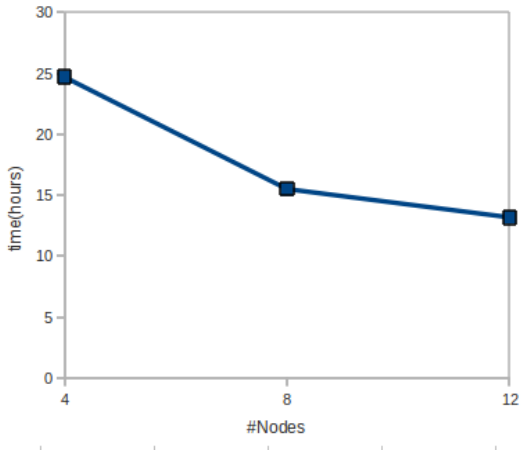

Figure 5.2: Number of compute Nodes Vs Time (Facility Location).

Data Vs Time

As we mentioned before the second type of our experiments checks the performance of our algorithms comparing the time needed to complete clustering in different-sized datasets. We expect our algorithms to be scalable with linear raise of time as the dataset will be getting bigger.

In Table below we see the results from K-Means algorithm on 12 working nodes and datasets of 300 MB ,600 MB and 1 GB.

	300 MB	**600 MB**	**1 GB**
12 nodes	3096104	5294255	7645795
$\sim$ hours	*0.86*	*1.47*	*2.1*

Table 5.30: K-Means , Datasets Comparing to Time

The figure below (5.3) summarizes the results above:

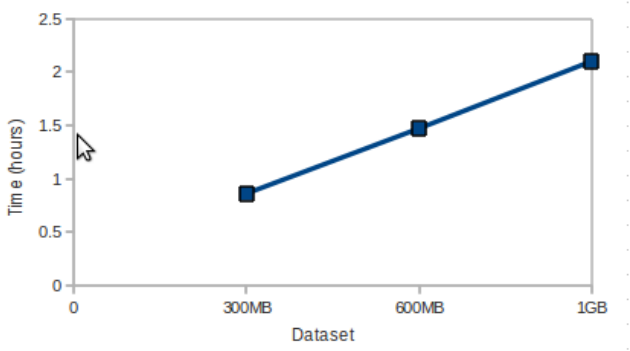

Figure 5.3: Datasets Vs Time (K-Means).

In figure 5.3 we see that our algorithm is working quite well as the time of completion is raising linearly to the size of datasets.

The same experiment has also been done for our second algorithm, Facility Location. In *FL* we expect to see also a linear graph but due to the bottleneck that algorithm faces in the first stage it would be normal to see the line start a bit higher than zero.

	300 MB	**600 MB**	**1 GB**
12 nodes	21229034	33859397	47461542
$\sim$ hours	*5.9*	*9.4*	*13.18*

Table 5.31: Facility Location , Datasets Comparing to Time

The figure below (5.4) summarizes the results above:

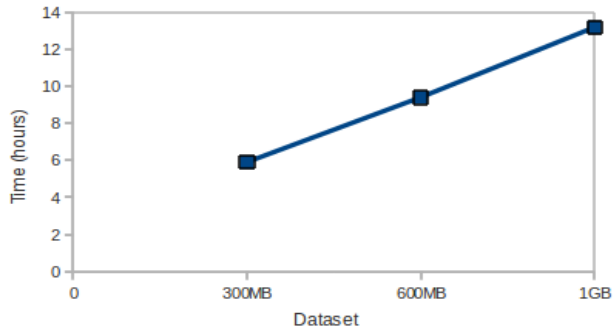

Figure 5.4: Datasets Vs Time (Facility Location).

Figure 5.4 depicts the problem we mention before. Although we have a constant raise of time, our algorithm needs a initial time limit that is wasted in

the first stage of our algorithm that is not so scalable.

Chapter 6

Conclusions

The motivation of our work has been to develop a parallel Map/Reduce algorithm in Hadoop Online Prototype to handle data in streams as they become available.

However, sequential algorithms cannot address the problem of data that occurs in real world networks. Hadoop is a tool that offers us the possibility to easily write parallel algorithms without caring about parallelization details like the communication of machines, the distribution of data, the replication and fault tolerance. All that is needed for a programmer, is to supply the implementation of a map and a reduce function. Hadoop is a powerful tool and has already been used in clustering algorithms like Fuzzy K-Means, Mean Shift, Dirichlet process clustering ([4]), K-Means ([12]) etc.

We worked on Hadoop Online Prototype (HOP), which is a modified version of Hadoop and allows data to be pipelined between tasks and between jobs. This can enable better cluster utilization and increased parallelism, and allows new functionality: *online aggregation* and *stream processing*. The contribution of our work is to supply two scalable algorithms that produce clustering on streams; K-Means algorithm and Local Search algorithm solving the Facility Location Problem. We performed several experiments in datasets with sizes from 13,000 to 80,000,000 multi-dimensional points and deducted interesting results. The experimental process has led us to the conclusion that the performance of our parallel algorithms is totally controlled by the I/O operations which are pretty heavy due to the great number of iterations, especially in dense datasets. However, we have proved through experimental studies that they scale up well and can be used in massive datasets, if we have a big cluster at our disposal.

Future work could focus on the effort of reducing the time of completion for each dataset by resolving the bottleneck problem that we are facing in the Facility Location. Furthermore, future work includes the development in Hadoop of more stream algorithms with more emphasis on pipelining functionality given by HOP and experiment on real streams.

Bibliography

[1] http://hadoop.apache.org.

[2] code.google.com/p/hop/wiki/HopConfiguration.

[3] kdnuggets.com/datasets/index.html.

[4] mahout.hadoop.org.

[5] Moses Charikar and Sudipto Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 378–, Washington, DC, USA, 1999. IEEE Computer Society.

[6] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. Technical Report UCB/EECS-2009-136, EECS Department, University of California, Berkeley, Oct 2009.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[8] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *IEEE Transactions on Knowledge and Data Engineering*, 15:515–528, 2003.

[9] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, SIGMOD '97, pages 171–182, New York, NY, USA, June 1997. ACM.

[10] Kamal Jain and Vijay V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. *J. ACM*, 48:274–296, March 2001.

[11] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[12] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on mapreduce. In Martin Jaatun, Gansen Zhao, and Chunming Rong, editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 674–679. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-10665-1_71.