DATA CLUSTERING USING MAPREDUCE

by

Makho Ngazimbi

A project

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

March 2009

The project presented by *Makho Ngazimbi* entitled *Data Clustering Using MapReduce* is hereby approved.

Amit Jain, Advisor Date

Gang-Ryung Uh, Committee Member Date

Tim Barber, Committee Member Date

John R. Pelton, Graduate Dean Date

Dedicated to my best friend Laura

ACKNOWLEDGEMENTS

Many thanks to my advisor Dr. Amit Jain. Thanks also to my committee members; Dr. Tim Barber and Dr. Gang-Ryung Uh. Finally I would like to thank my family and friends without whose love and support I could not have completed this long and difficult endeavor.

AUTOBIOGRAPHICAL SKETCH

I am a native of Bulawayo, a large city in south west Zimbabwe. I came to Pocatello, Idaho in 2002 and received my Bachelor of Computer Science in 2006. I moved to Boise, Idaho in 2007 to pursue my Master of Computer Science, which I hope to receive in 2009. My main areas of interest in computing are databases, parallel computing, and distributed computing. In my free time I enjoy playing soccer at the various indoor leagues around Boise, and I play field hockey with Les Bois sports organization.

ABSTRACT

MapReduce is a software framework that allows certain kinds of parallelizable or distributable problems involving large data sets to be solved using computing clusters. MapReduce is attractive because it abstracts parallel and distributed concepts in such a way that it allows novice programmers to take advantage of cluster computing without needing to be familiar with associated complexities such as data dependency, mutual exclusion, replication, and reliability. However, the challenge is that problems must be expressed in such a way that they can be solved using MapReduce. This often involves carefully designing inputs and outputs of MapReduce problems as often outputs of one MapReduce are used as inputs to another.

Data clustering is a common computing task that often involves large data sets for which MapReduce can be an attractive means to a solution. This report presents a case study of clustering Netflix movie data using K-means, Greedy Agglomerative, and Expectation Maximization clustering algorithms using Apache Hadoop MapReduce framework. Netflix is a large online DVD rental service. A major part of Netflix's revenue generation can be directly attributed to providing movie recommendations to customers based on movies they have seen and rated in the past. As part of an ongoing effort to improve this movie recommendation system, Netflix is sponsoring a competition for the best movie rating predictor. Netflix provides a data set containing over 100 million ratings to competition participants, which we use in our MapReduce clustering case study. Root Mean Square Error (RMSE) is used to compare actual ratings versus predictions made on this data set. Netflix's predictor achieves a RMSE of 0.9525 on the provided data set. A demonstration is provided on how the resulting clustered data can be used to create a simple movie rating predictor that is able to achieve a RMSE of 1.0269 which is within 8% of Netflix's predictor. The predictor demonstrated also highlights an interesting collaboration of MapReduce and Database Management Systems, namely Hadoop and MySQL.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Data clustering [1] is the partitioning of a data set or sets of data into similar subsets. During the process of data clustering a method is often required to determine how similar one object or groups of objects is to another. This method is usually encompassed by some kind of distance measure. Data clustering is a common technique used in data analysis and is used in many fields including statistics, data mining, and image analysis. There are many types of clustering algorithms. Hierarchical algorithms build successive clusters using previously defined clusters. Hierarchical algorithms can be agglomerative meaning they build clusters by successively merging smaller ones, which is also known as bottom-up. They can also be divisive meaning they build clusters by successively splitting larger clusters, which is also known as top-down. Clustering algorithms can also be partitional meaning they determine all clusters at once.

Data clustering can be computationally expensive in terms of time and space complexity. In addition further expense may be incurred by the need to repeat data clustering. Hence, parallelizing and distributing expensive data clustering tasks becomes attractive in terms of speed-up of computation and the increased amount of memory available in a computing cluster. Programming distributed memory systems Message Passing Interface (MPI) is a widely used standard. A disadvantage of MPI is that the programmer must have a sophisticated knowledge of parallel computing concepts such as deadlocks and synchronization.

MapReduce is a software framework for solving certain kinds of distributable problems using a computing cluster. In its simplest form MapReduce is a two step process. In the Map step a master node divides a problem into a number of independent parts that are assigned to map tasks. Each map task processes its part of the problem and outputs results as key-value pairs. The reduce step receives the outputs of the maps, where a particular reducer will receive only map outputs with a particular key and will process those. The power of MapReduce comes from the fact that Map and Reduce tasks can be distributed across different nodes. Hence, by design MapReduce is a distributed sorting platform. Since Maps are independent they can be run in parallel similarly to reduce tasks which can complete after maps complete.

Apache Hadoop [2] is a free Java MapReduce framework that allows a novice parallel or distributed programmer to utilize a computing cluster without having to understand any of the associated concepts. Hadoop was inspired by Google's MapReduce [3] and Google File System (GFS) [4] papers. Hence, the minimally skilled Hadoop programmer must be familiar with the Java programming language, able to express a software problem in MapReduce terms, and understand the concept of a distributed file system. Hadoop is an attractive distributed computing framework for many reasons. These include reliability achieved by replication, scales well to thousands of nodes, can handle petabytes of data, automatic handling of node failures, and is designed to run well on heterogeneous commodity class hardware clusters. However, Hadoop is still a fairly new project and limited example code and documentation is available for non-trivial applications.

This report presents a case study of clustering Netflix [5] movie data using Hadoop thereby achieving a non-trivial collection of sample code and documentation. Netflix is a large online DVD rental service in the United States. Netflix began a contest in October 2006 known as Netflix Prize [6] for the best collaborative filtering [7] algorithm that makes user predictions on films based on previous predictions. The competition is expected to last at least until October 2011 at which point the leading algorithm will receive a large cash prize if it can outperform Netflix's current predictor by a certain margin. Netflix provides contestants with a data set consisting of 100 million ratings from 480,000 users, made on a set of over 17,000 movies. Each user rates a movie on a scale of one to five. This data set presents an interesting case study for a data clustering implementation using MapReduce.

Three common data clustering algorithms implemented in this study are K-means [8], Greedy Agglomerative [9], and Expectation-Maximization [10]. These algorithms are computationally expensive and perform well in data sets with the following three characteristics: relatively low feature dimensionality, limited number of clusters, and a small number of data points. The Netflix data however, is large in all three areas hence processing this data can be expected to be a very computationally expensive task. In this study we explore a novel concept know as canopy clustering [11] where data is first clustered into overlapping canopies using a comparatively cheap distance measure, then the data is clustered further using the expensive aforementioned-mentioned clustering algorithms. Canopy clustering has been proven not to reduce the clustering accuracy but instead improve computational efficiency since data points need only be compared to others belonging to the same or overlapping canopies instead of making comparisons to every single point in the data set. In fact with a large data set such as the Netflix problem it would be impractical to use any of these clustering algorithms as physical memory and computational speed would become limiting factors.

The most obvious application of this particular data clustering is for use in predicting ratings since that is the goal of the Netflix prize. Predictions can be made based on a user's previous ratings and those of users who have rated movies belonging to the same clusters. Another possible application of this data clustering is to recommend movies to a user based on others they have previously seen and those seen by other users in the same clusters. An example rating predictor is demonstrated, which is implemented in MapReduce. This simple predictor also demonstrates the collaboration of Hadoop and MySQL, which provides a feasible solution to a reliable, stable, resilient, and scalable source of input and output for MapReduce jobs.

Chapter two explains the various aforementioned clustering algorithms that we implement using MapReduce. Chapter three discusses Hadoop which is the MapReduce implementation we used. In this chapter we also detail the MapReduce strategy we use to tackle this particular data clustering problem. In chapter four we see how the resulting clustered data can be used to create a movie rating predictor, and a web based movie recommendation system. Chapter five summarizes the report and provides details on future directions.

Chapter 2

DATA CLUSTERING

2.1 Overview

Data clustering is the partitioning of object into groups (called clusters) such that the similarity between members of the same group is maximized and similarity between members of different groups is minimized. Often some form of distance measure is used to determine similarity of objects. See Section 2.2 for details on distance metrics used in this project. There are several types of clustering algorithms.

Hierarchical algorithms find successive clusters using previously determined clusters. Hierarchical algorithms can be either agglomerative (bottom-up) or divisive (top-down). Agglomerative algorithms begin with each object as singleton clusters and successively merge those with other clusters to create the final clusters. Divisive algorithms start with the entire data set of objects and partition the set successively into smaller clusters.

Partitional algorithms typically determine all clusters at once. Often partitional algorithms can be used as divisive algorithms in divisive hierarchical algorithms.

2.2 Distance Metrics

2.2.1 Cosine Similarity

Cosine similarity is a measure of the distance between two vectors of dimension n, by finding the cosine of the angle between them. It is often used in text mining to compare documents.

Given two vectors A and B the cosine similarity is given by the formula:

$$similarity = \cos(\theta) = \frac{A \cdot B}{||A|| \cdot ||B||}$$

The resulting similarity ranges from -1 meaning exactly opposite, to 1 meaning exactly the same, 0 indicating independence. Generally the closer the result to one the more similar the vectors.

2.2.2 Mahalanobis Distance

Mahalanobis distance is a useful method of determining the similarity of an unknown sample set to a known set. It differs from Euclidean distance in that it takes into account the correlations of the data set and is not dependent on the scale of measurements.

The Mahalanobis distance from a group of values with mean $\mu = (\mu_1, \mu_2, \mu_3, \ldots, \mu_N)^T$ and covariance matrix S for a multivariate vector $x = (x_1, x_2, x_3, \ldots, x_N)^T$ is defined as:

$$D_M(x) = \sqrt{(x - \mu)^T S^{-1} (x - \mu)}$$

Mahalanobis distance is widely used in cluster analysis and other classification techniques. In order to classify a data point as belonging to one of N classes, the covariance matrix of each class must be computed using known data points. Given a test point the Mahalanobis distance is computed to each class and the test point is assigned to the point with minimal distance. This is equivalent to selecting the class with maximum likelihood.

2.3 Canopy Clustering

Canopy clustering is an algorithm intended for combined use with another clustering algorithm whose direct usage may be impractical due to the large size of a data set. Using canopy clustering the data is first partitioned into overlapping canopies using a cheap distance metric. The data is then clustered using more traditional clustering algorithms such as K-means which are usually more costly due to more expensive distance measures. Canopy clustering allows these expensive distance comparisons to be made only between members of a particular canopy rather than across all members of the data set. Under reasonable assumptions of the cheap distance measure it is possible to reduce computational time over a traditional clustering algorithm without a significant loss in clustering accuracy. Hence, canopy clustering can be considered a divide-and-conquer heuristic in the sense that it allows an optimal solution to be reached rapidly which may or may not be the optimum solution. However, the divide-and-conquer aspect of canopy clustering may be essential in very large data sets where the space and time complexity of traditional clustering algorithms across a large number of data points with a high feature dimensionality may prove impossible to solve due to hardware memory limitations especially where high computational speed is desirable.

2.4 K-means Clustering

K-means is a partitional clustering algorithm that partitions n objects into some chosen k number of clusters, where $k < n$. The most commonly implemented form of the algorithm uses an iterative refinement technique known as Lloyd's heuristic. In fact Lloyd's heuristic is so commonly used with K-means clustering that it is often mistakenly described as K-means clustering when in fact it is just a heuristic. Lloyd's algorithm begins by partitioning data into k sets using some defined method or even arbitrarily. The mean point or centroid of each set is then calculated and the algorithm is repeated by associating each data point to the nearest centroid then finding the new set center. This is done repeatedly until convergence, which is determined by observing that the centroids no longer change upon successive iterations. As with any heuristic Lloyd's algorithm does not guarantee a globally optimal solution and can in fact converge to the wrong answer. Other heuristics and even variants of Lloyd's heuristic [12] exist but the basic Lloyd's heuristic is popular because in practice it converges very quickly. However, it has been demonstrated that in some situations Lloyd's heuristic can converge in superpolynomial time [13]. Since the algorithm is usually extremely fast, a common practice is to run the algorithm several times and return the best clustering found. A disadvantage of the K-means algorithm is that the value of k is an input parameter and an inappropriate value may yield poor results.

2.5 Greedy Agglomerative Clustering

Agglomerative clustering builds the desired clusters from single data objects. A Greedy approach is to merge the two clusters with the greatest similarity at each step. This process is repeated until either the desired number of clusters is achieved or until the resulting clusters all meet some predefined characteristic. The following figure demonstrates an agglomerative clustering. In the example we have six singleton

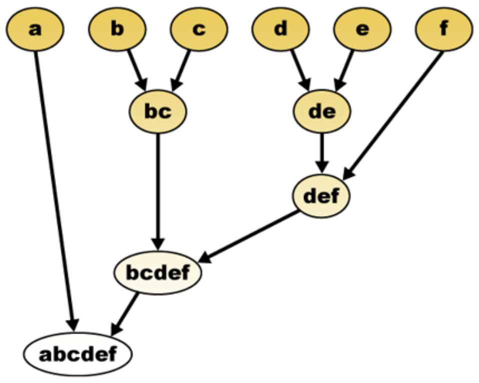

Figure 2.1. Progressive merging of clusters

clusters {a}, {b}, {c}, {d}, {e}, and {f}. The first step is to determine which object to merge into a cluster. In the Greedy approach the clusters can be merged based on which are closest to each other based on the distance measure. For example in the first round of mergings, the new clusters {b,c} and {d,e} may be formed. This continues until the final cluster {a,b,c,d,e,f} has been created.

2.6 Expectation-Maximization Clustering

Expectation-Maximization (EM) algorithm is an iterative technique for estimating the value of some unknown quantity, given the values of some correlated, known quantity. EM is generally preferable to K-means due to its better convergence properties. EM is popular in statistical estimation problems involving incomplete or hidden data [14]. The EM procedure [15] is:

1. Initialize the distribution parameters,
2. Repeat until convergence:
    (a) E-Step: estimate the [E]xpected value of the unknown variables, given the current parameter estimate,
    (b) M-Step: re-estimate the distribution parameters to [M]aximize the likelihood of the data, given the estimates of the expectations of the unknown variables.

These steps of the EM algorithm depend on the distribution parameters chosen, how many parameters there are, and how complicated the missing value is. In addition the choice of initial parameters is important because the algorithm may only converge to a local optimum. In practice, poor choices can lead to bad results. Although convergence is guaranteed it may take a long time to achieve. In practice, if the missing variables and parameters do not change significantly between successive iterations then the algorithm terminates.

EM algorithm applied to data clustering is very similar to the iterative K-means algorithm. The main points of EM clustering are:

1. Initial cluster centers are selected. Membership to a cluster is determined by a probability. For each point there are as many probabilities as there are clusters. For each point the sum of probabilities equals one.

2. Clusters are defined by a cluster center and a cluster covariance matrix. Cluster centers and the covariance matrix determine a Mahalanobis [16] distance between a cluster center and a point.

3. Cluster probabilities, cluster centers and covariance matrices are recalculated iteratively. In the E-step, for each cluster the probabilities are calculated.

4. In the M-step all cluster centers and covariance matrices are recalculated from the updated probabilities, so that the resulting data likelihood function is maximized.

5. When the iteration is completed, each point is assigned to the cluster where the probability is maximal. In practice this is equivalent to assigning the point to the cluster where the Mahalanobis distance is the least.

Chapter 3

MAPREDUCE AND HADOOP

3.1 Hadoop

3.1.1 Overview

MapReduce is a framework that allows certain kinds of problems particularly those involving large data sets to be computed using many computers. Problems suitable for processing with MapReduce must usually be easily split into independent subtasks that can be processed in parallel. In parallel computing such problems as known as embarrassingly parallel and are ideally suited to distributed programming. MapReduce was introduced by Google and is inspired by the `map` and `reduce` functions in functional programming languages such as Lisp. In order to write a MapReduce program, one must normally specify a mapping function and a reducing function. The `map` and `reduce` functions are both specified in terms of data the is structured in key-value pairs. The power of MapReduce is from the execution of many map tasks which run in parallel on a data set and these output the processed data in intermediate key-value pairs. Each reduce task only receives and processes data for one particular key at a time and outputs the data it processes as key-value pairs. Hence, MapReduce in its most basic usage is a distributed sorting framework. MapReduce is attractive because it allows a programmer to write software for execution on a computing cluster with little knowledge of parallel or distributed computing. Hadoop is a free open source MapReduce implementation and is developed by contributors from across the world. The version used in this project is 0.18.1. Hadoop scales well to many thousands of nodes and can handle petabytes of data. The Hadoop architecture has two main parts: Hadoop Distributed File System (HDFS) and the MapReduce engine. Basically HDFS can be used to store large amounts of data and the MapReduce engine is capable of processing that data.

A related sub-project of Hadoop, HBase database which runs on top of HDFS has arisen out of the need to process very large databases in the petabyte range. HBase is a column oriented distributed database based on Google's Big Table [17] and is designed to run on top of Hadoop providing input and output for MapReduce jobs. This can be compared to pairing Hadoop with a common database management system such as MySQL [18].

3.1.2 Hadoop Distributed File System (HDFS)

HDFS is a pure Java file system designed to handle very large files. Similar to traditional filesystems, Hadoop has a set commands for basic tasks such as deleting files, changing directory and listing files. The files are stored on multiple machines and reliability is achieved by replicating across multiple machines. However, this is not directly visible to the user it and appears as though the filesystem exists on a single machine with a large amount of storage. By default the replication factor is 3 of which at least 1 node must be located on a different rack from the other two. Hadoop is designed to be able to run on commodity class hardware. The need for RAID storage on nodes is eliminated by the replication.

A limitation of HDFS is that it can not be directly mounted onto existing operating systems. Hence, in order to execute a job there must be a prior procedure that involves transferring the data onto the filesystem and and when job is complete the data must be removed from the filesystem. This can be an inconvenient time consuming process. One way to get around the extra cost incurred by this process is to maintain a cluster that is always running Hadoop such that input and output data is permanently maintained on the HDFS.

3.1.3 MapReduce Engine

The Hadoop mapreduce engine consists of a job tracker and one or many task trackers. A mapreduce job must be submitted to a job tracker which then splits the job into tasks handled by the task trackers. HDFS is a rack aware filesystem and will try to allocate tasks on the same node as the data for those tasks. If that node is not available for the task, then the task will be allocated to another node on the same rack. This has the effect of minimizing network traffic on the cluster back bone and increases overall efficiency.

In this system the job tracker becomes a single point of failure since if the job tracker fails the entire job must be restarted. As of Hadoop version 0.18.1 there no task checkpointing so if a tasks fails during execution is must be restarted. Hence, it also follows that a job is dependent on the task that takes the longest time to complete since all tasks must complete in order for the job to complete.

3.2 The Netflix Problem

An important part of Netflix's revenue generation comes from recommending movies to users based on movies they have seen in the past. As part of an effort to improve their movie recommendation system, Netflix is currently sponsoring the Netflix Prize. This is a competition that began in October 2006 and is expected to run at least until October 2011. The winning entry will be an algorithm that is able to predict user ratings for movies most accurately. In order to qualify for the grand prize the best algorithm must have a Root Mean Square Error (RMSE) that is 10% better than Netflix's current predictor. RMSE is used to compare how much the actual ratings made by users vary from predictions. A low RMSE indicates that a predictor is more accurate than one with a higher RMSE. Actual ratings and predictions for a particular movie by certain raters can be represented as vectors. Figure 3.1 shows an example.

The formula for the RMSE is given by:

$$RMSE(\theta_1, \theta_2) = \sqrt{MSE(\theta_1, \theta_2)} = \sqrt{E((\theta_1 - \theta_2)^2)} = \sqrt{\frac{\sum_{i=1}^{n}(x_{1,i} - x_{2,i})^2}{n}}$$

$$\theta_1 = \begin{bmatrix} x_{1,1} \\ x_{1,2} \\ \vdots \\ x_{1,n} \end{bmatrix} \qquad \text{and} \qquad \theta_2 = \begin{bmatrix} x_{2,1} \\ x_{2,2} \\ \vdots \\ x_{2,n} \end{bmatrix}.$$

Figure 3.1. Two vectors representing data points

Netflix provides competition entrants with a data set that includes 100 million ratings from over 480,000 users and more than 17,000 movies. The files containing the ratings are called the training set. Raters are anonymous and are identified only by numbers which are hereafter referred to as *raterIDs*. Movies are identified by numbers hereafter referred to as *movieIDs*. A file is also provided that matches *movieIDs* to movie titles. A *probe* file is also provided that contains 1.4 million *movieID* and *raterID* pairs for which predictions must be made. All the actual ratings for the movies and raters in the *probe* file are contained in the training set. Cinematch, which is Netflix's current predictor is able to achieve a RMSE of 0.9525 on this probe set. So this can be used as a benchmark for evaluating a predictor.

3.3 MapReduce Strategy

3.3.1 Overview

The greatest challenge to processing and solving a problem using Hadoop's MapReduce system is designing the inputs and outputs. Complex problems such as the one being considered in this report must often be performed in multiple MapReduce steps. Each step takes as input the output from a previous step MapReduce. The Netflix data clustering consists of five steps outlined in this section.

3.3.2 Step 1 - Data Preparation

This step transforms the original Netflix provided movie data set into a form that can be more easily processed using Hadoop. This data set is a collection of more than 17,000 text files each containing data for a single movie. Each of these files contains the movie identification number of the movie as the first line. Every subsequent line contains comma separated values for a rater identification number, an integer rating greater than or equal to one and less than or equal to five given by that rater, and the date on which the movie was rated. For example the first five lines of the movie "The Name of the Rose" with movie identification number 11,674 would look like this:

```
11674:
1331154,4,2004-08-02
551423,5,2004-07-19
716091,4,2005-07-18
```

Since by default Hadoop mappers process input text files line by line, it is logical to transform each file into a single line such that each movie can be processed by a single map. Hence, the final output maybe a collection of text files each with the format:

movieID_D `rater_i:rating_i,rater_j:rating_j,rater_k:rating_k,...`

```
movieID_E rater_u:rating_u,rater_v:rating_v,rater_w:rating_w,...
...
```

For example the data for a the movie "The Name of the Rose" will be transformed into the format:

```
11674 1331154:4,551423:5,716091:4,1174530:3,...
```

Map Step

The intermediate output of the map will be movieID as the key and raterID as the value.

Reduce Step

The output of the reduce step will simply output movieID as the key and concatenate the raterIDs for that movie into a comma separated list.

3.3.3 Step 2 - Canopy Selection

In this step the canopy centers are chosen using a "cheap" distance metric. The metric chosen is described as follows: if a set of z number of people rate movie A and the same set of z number of people rate movie B, then movies A and B belong to the same canopy. Using this metric it is possible that canopies may overlap, or in other words a movie may belong to multiple canopies. So long as each movie belongs to at least one canopy the necessary condition of canopy clustering will be met. Hence, in order for this to be true the value z must not be too large as the canopies may be large and many data points may lie outside of canopies. If the value of z is too small then the number of canopies will be few and each canopy will have many data points. Hence, the eventual expensive data clustering may not be very good. During testing the value of z used was 10 resulting in about 400 canopies. Table 3.1 shows the effect of varying the distance on the number of canopies produced.

The input used for this step is the output from Step 1. Hence each line from the text input files has the format:

```
movieID_D rater_i:rating_i,rater_j:rating_j,rater_k:rating_k,...
```

The output will be a file containing the canopy centers along with their rating data. Since it is a subset of the full data set it has the same format as the input files.

Map Step

Every mapper maintains a collection containing the canopy center candidates it has determined thus far. During every map the mapper determines if each successive movie is within the distance threshold of any already determined canopy center candidate. If the mapped movie is within the threshold then it is discarded, otherwise it is added to the collection of canopy center candidates. The intermediate output sent to the reducer has the movieID as the key and the list of raterID-rating pairs as the value.

Reduce Step

The reducer repeats the same process as the mappers. It receives the candidate canopy center movieIDs but removes those which are within the same threshold. In other words it removes duplicate candidates for the same canopy center. In order for this to work correctly the number of reducers is set to one.

TABLE 3.1 Effect of distance on number of canopies created

Distance	Canopy Centers
5	75
6	108
7	169
8	251
9	326
10	433
11	530
12	685

3.3.4 Step 3 - Mark By Canopy

This step marks each movie from the full data set from Step 1 with the identification number of the canopies it belongs to. The two inputs used for this step are the output from Step 1 and the output from Step 2. The same distance metric from Step 2 is used in this step to determine if the movie belongs to a particular canopy or not. The output will have the following format:

```
movie_A rater_i:rating_i,rater_j:rating_j,... ;canopy_U,canopy_V,...
```

Map Step

Each mapper will load the canopy centers generated by Step 2. As each movie is received from the full data set the mapper determines the list of canopy centers that the movie is within, using the same distance metric from Step 2. The intermediate output is the movieID as the key and its raterID-rating pairs and list of canopies as the value.

Reduce Step

The reducers simply output the map output.

3.3.5 Step 4 - Expensive Clustering

The expensive clustering steps do not change the full movie data set. They merely move around the canopy centers so a more accurate clustering is determined. The details for each clustering type are explained in this section. `SequenceFile` output format is used for all MapReduces in this step. In this step we found it necessary to avoid Java's `java.lang.OutOfMemoryError` error by limiting the number of ratings considered per movie to 150,000. In addition the maximum number of movies per cluster to consider for the new cluster center in this step is limited to the top 1,000 most likely. These are chosen from the set of movies most similar to the current canopy center.

K-means Clustering

The K-means clustering step is performed iteratively until convergence is achieved. In simple terms this means until the k-centers no longer change. However, in practice this can take an incredible amount of time or never be achieved at all. So for testing purposes the algorithm was run iteratively up to five times and the final result considered converged. The expensive distance metric used in this step is cosine similarity. The two input data sets for this step are the full data sets marked with canopies created by Step 3 and initially the canopy centers created by Step 2. The output is a list with the new cluster centers (movieID) as the key and it raterID-rating pairs list as its values in the same format as the output of the canopy selection MapReduce (Step 2).

Map Step

Each mapper loads the k-centers from the previous K-means MapReduce into memory. For the first iteration the canopy centers generated by Step 2 are used. Each movie that is mapped is also contains a list of the canopies it belongs to. Using the expensive distance metric the mapper determines which canopy the movie is closest to and outputs the chosen canopyID as the key and the mapped movie as the value.

Reduce Step

The reduce step must determine the new center for every canopyID that it receives from the mappers. The process to do this involves determining the theoretical average movie in a canopy, then finding the actual movie that is most similar to this average value. When finding the average movie one determines the set of raters that belong to a canopy. For each of these raters it can be determined how they scored movies on average. With this information we can use cosine similarity to determine the movie most similar to this average movie.

Greedy Agglomerative Clustering

This clustering can be performed in a single MapReduce. The canopy centers created by Step 2 and the full movie data set marked by canopyID from Step 3 are used as the inputs. The canopy centers are used as the starting points for each eventual cluster that would be created by the Greedy Agglomerative Clustering algorithm. Cosine similarity is the distance metric used in this step. In this step when a movie is mapped, its membership to a cluster is determined not by its proximity to the cluster center but by its proximity to the current average data point in the cluster. When a desired cluster strength, or number of movies is achieved in a cluster then no more movies can be added to that cluster.

Map Step

Each map loads the canopy centers into memory. As movies are mapped, a canopy center is chosen from the list the movie belongs to provided that canopy has not already reached its desired size. The intermediate output of this step is the chosen canopy center as the key and movie data as the value.

Reduce Step

The reduce step determines the new canopy center using the same method as the reduce step of the K-means MapReduce.

Expectation-Maximization Clustering

The MapReduce steps for the Expectation-Maximization (EM) clustering are very similar to those for the the K-means algorithm. In fact EM clustering can be considering a generalized (although more complex) K-means clustering. EM clustering is also iterative like K-means. The Mahalanobis [19] distance measure is used in this step. The input to this step is initially the canopy centers created by Step 2 and the full data set marked by canopy centers from Step 3.

Map Step

The maps encompass the Expectation step of the EM algorithm where the cluster probabilities are calculated. In this case, however, the maximum likelihood function is calculated and the minimum value is selected which is equivalent to selecting the canopy with the greatest probability. Cluster membership is determined similarly to the K-means clustering MapReduce by comparing mapped movies to the current cluster center using the Mahalanobis distance metric.

Reduce Step

The reduces encompass the Maximization step of the EM algorithm where new cluster centers are calculated similarly to the K-means step.

3.3.6 Step 5 - Inverse Indexer

This phase outputs results that can be used. The aim is to associate each movie with a cluster center in an inverted index format. Hence, the cluster center movie identification numbers are used as the keys and the associated movie identification numbers are used as the values. The two inputs for this step are the list of centroids and the full movie data set. The output has the following format:

```
movieID_centroid movieID_A:similarityA,movieID_B:similarityB,...
```

Map Step

The map loads the cluster centers determined by any one of the algorithms from Step 4. For each mapped movie that is within the cheap distance metric from Step 2 of any cluster center, the similarity is calculated for that movie to the cluster center using the appropriate distance metric. The intermediate output sent to the reducer will have the cluster center as the key and the mapped movieID and similarity as the value.

Reduce Step

The reduce simply concatenates the movieID-similarity pairs for each cluster center.

Chapter 4

NETFLIX SPECIFIC APPLICATIONS OF DATA CLUSTERING

4.1 Predictor for Netflix Prize

A simple predictor of Netflix ratings given a specific user and movie can be created using the clustered data. The accuracy of this predictor can be compared to that of Cinematch, which is the name of Netflix's current predictor. Netflix provides a "probe" file containing over 1.4 million userID and movieID pairs for each of which a prediction in the range one to five must be made. The rating for each of these pairs is actually contained in the dataset provided by Netflix so it is possible to make a prediction and retrieve the actual value for comparison purposes. Hence, an example predictor using this probe file for input can produce as output 1.4 million actual rating - predicted rating pairs. This output can be used to compute a Root Mean Square Error (RMSE) [20] value for comparison purposes against Cinematch, which is able to achieve a RMSE of 0.9514 on this probe data.

4.1.1 Random Predictor

The purpose of this MapReduce is to produce a simple Random Predictor for comparison purposes. Since the ratings for all the movieID-raterID pairs in the "probe" file are contained in the full data set, the Random Predictor MapReduce can retrieve those actual ratings and we can use those for analytical purposes. The input to this MapReduce is the full data set and the "probe" file of movieIDs and raters for which to make predictions. The probe file has the format:

```
movieID1 raterA,raterB,raterC...
movieID2 raterX,raterY,raterZ...
...
```

The output will have the format:

```
<actual rating>,<predicted rating>
<actual rating>,<predicted rating>
...
```

where each line represents the ratings made by a user and predicted for a user for a particular movie.

Map Step

Since the probe file contains ratings already present in the full data set, the map retrieves the actual rating made by a particular user for a certain movie. The intermediate output sent to the reduce is keyed by movieID and raterID, and the value is the rating.

Reduce Step

The reduce should only receive a single actual rating value for every movieID-rater pair. The reduce then makes a random prediction between and including the integers one to five.

4.1.2 Predictor Data Preparation

The purpose of this step is to produce a list of a few similar movies for every movie and rater pair that we are interested in making a prediction for. Given a collection of similar movies there will be a larger pool of users and ratings from which to base a prediction for a particular user.

This step highlights the collaboration of MySQL and Hadoop. In this step the map opens the connection to the database in its configure method call. The connection is explicitly closed when the map completes by overriding the close method to do so. A mistake would be to open the connection in the actual map call which we did and caused extremely poor performance as we quickly exhausted the number of database connections available to me. Opening the connection in the configure call is also preferable because then it is done once by every map instead of once for every line of input mapped. Hence, for a particular job 120 connections would be made corresponding to 120 maps instead of 1.4 million connections corresponding to 1.4

million input lines of map input (or ratings to be made). The input to the map is the probe file. The output will have the format:

```
probeMovieID similarMovie1,similarMovie2,...
```

Map Step

The database is queried to find several similar movies using the following procedure:

1. If the movie is a cluster center then fetch several of the most similar to it.
2. If the movie is not a cluster center:
   (a) Find the cluster center it is closest to.
   (b) Fetch several movies most similar to this cluster center.

The intermediate output sent to the reduce is keyed by movieID and the value is a list of the similar movies. The database schema are available in Appendix C.

Reduce Step

The reduce simply concatenates the similar movies for a particular probe movie.

4.1.3 Weighted Mean Predictor

This simple predictor make predictions based on the following three input parameters that provide a weighted contribution to the prediction:

1. α, the mean of ratings a particular user has made in the past.

2. β, the mean of ratings that similar raters have made in the past.

3. E[Movie Rating], the expected value of a rating for a movie based on observation of how Netflix user's usually rate movies.

The expected value for a movie rating across the data set is calculated using the formula:

$$E[MovieRating] = \sum_i x_i p(x_i)$$

I determined the expected rating by observing the occurrence of each rating in the training set. The results are shown in Table 4.1.

TABLE 4.1 Distribution of ratings in Netflix training data set

Rating	Frequency	Percentage
1	73,211	5.2%
2	136,081	9.7%
3	352,436	25.0%
4	462,093	32.8%
5	384,573	27.3%
Total	1,408,394	100.0%

Applying the formula results in the calculation:

$$E[MovieRating] = 1(0.052) + 2(0.097) + 3(0.25) + 4(0.328) + 5(0.273)$$

Hence, the expected rating for a movie based on calculations made from all movies in the training set is `3.6`. Figure 4.1 shows a graphical representation of the percentage occurrence for each of the ratings.

For experimental purposes the formula used to make a prediction is:

$Prediction = (\alpha * 0.3) + (\beta * 0.6) + (E[MovieRating] * 0.1)$

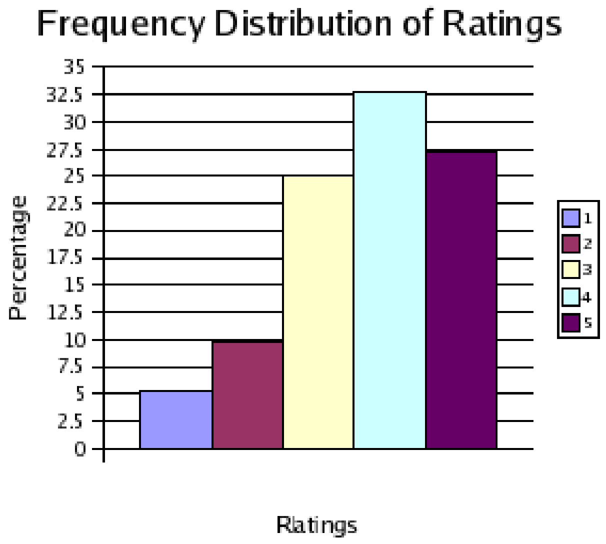

Figure 4.1. Frequency distribution of ratings

The input for this MapReduce is the full data set and the similar movies list created by the Predictor data preparation MapReduce. The output is actual rating - predicted rating pairs for every movie-rater combination, and is of the format:

```
<actual rating>,<predicted rating>

<actual rating>,<predicted rating>

...
```

Map Step

The map loads the data from the predictor preparation step. As each map receives a movie from the full data set it determines if that movie is needed to make a prediction. This depends on wether it is a movie for which a prediction is needed or wether it is a similar movie. The intermediate output sent to the reducer is keyed by the movieID-rater pair and its list of similar movies. The value is the mapped movie and its raters and rating data.

Reduce Step

The reduce determines the actual ratings which will eventually be output and also makes the prediction for the movie. In this step we found it is necessary to limit the number of ratings per movie used to make a prediction in order to avoid Java's `java.lang.OutOfMemoryError` error. In our implementation we found it necessary to temporarily store the ratings for a movies prior to making a prediction. These can be a very large amount and we found that it is possible to store up to 150,000 per movie before running out of memory. In this step we also limited the raters considered to those within the 10% of those rating the most number of movies similar to the one we would like to predict a rating for.

Table 4.2 shows the RMSE values produced by the Predictor for the various clusterings. K-means and Expectation-Maximization are iterative algorithms so the results are shown for several iterations. In Table 4.2 we see that the best performance is is achieved using the K-means clustered data. Figure 4.2 shows a chart of the RMSE value produced by each algorithm after each iteration. From this chart we see that K-means performs best overall followed by Greedy Agglomerative, and then Expectation-Maximization. Generally Expectation-Maximization is used when better

TABLE 4.2 RMSE results of Predictor for the various clustering algorithms.

Iterations	K-means RMSE	Greedy Agglm. RMSE	Expect-Max. RMSE
1	1.02692	1.03780	1.10826
2	1.02690	N/A	1.08440
3	1.02694	N/A	1.08628
4	1.02691	N/A	1.08212
5	1.02693	N/A	1.08082

clustering than K-means is required. K-means usually produces better clustering than Greedy Agglomerative. Hence the results are not as expected. We would have expected Expectation-Maximization to produce the best results, followed by K-means, then Greedy Agglomerative. However, the performance of Expectation-Maximization clustering is largely determined by the input parameters particularly the initial clustering centers. K-means is also prone to this effect particularly for the initial number of K-centers chosen but not as much as Expectation-Maximization. This may explain the poor results for Expectation-Maximization. However, for the input parameters used in this experiment we see in Figure 4.2 that upon successive iterations of the algorithm the RMSE seems to improve. Eventually after enough iterations the RMSE may converge to a more acceptable value but the time cost due to the number of iterations may be unacceptable in practice. K-means quickly converges to a good RMSE after even one iteration so in this particular case it is very appealing. More experimentation would be required to finely tune the Expectation-Maximization parameters and achieve the best RMSE.

Table 4.3 shows the results of some comparisons of various predictors against Cinematch. The Simple Mean predictor shown is a trivial solution where the prediction

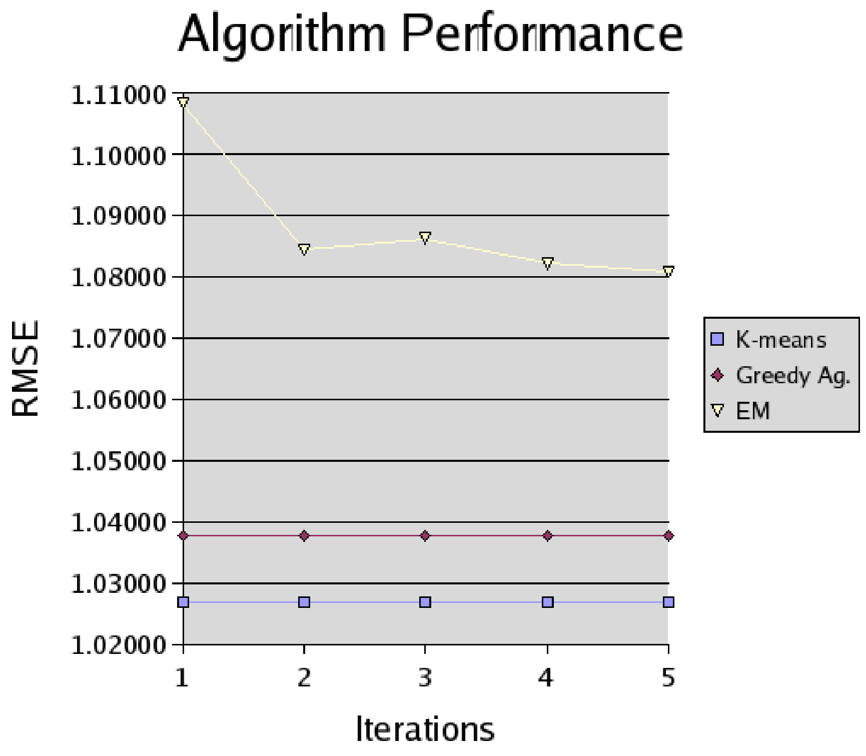

Figure 4.2. RMSE versus algorithm iteration chart

made for a movie is the mean of scores made by users who have rated that particular movie. In theory data clustering algorithms implemented should allow a more accurate prediction to be made by selecting only scores from appropriately similar raters to be used to make predictions rather than all scores. This seems to be the case particularly for the K-means and Greedy Agglomerative clustered data predictions, since there is an improvement over just averaging all scores. As expected a Random predictor performs very poorly compared to the other predictors. Pragmatic Theory is the name of the leading team in the Netflix Prize on April 28, 2009, who have already managed a 9% RMSE improvement on Cinematch. In order to qualify for the

TABLE 4.3 Selected RMSE comparisons versus Cinematch.

Method	RMSE	vs. Cinematch
Random	1.9487	-104%
Expectation-Maximization	1.0808	-13.5%
Simple Mean	1.0540	-10.7%
Greedy Agglomerative	1.0378	-8.96%
K-means	1.0269	-7.81%
Cinematch	0.9525	0.00%
Pragmatic Theory (April 28, 2009)	0.8596	+9.02%
Goal	**0.8572**	**+10.0%**

grand prize of one million dollars, the leading entry must achieve a RMSE of 0.8596 which would be a 10% improvement over Cinematch. Obviously a good predictor must also take into account other data in order to be more accurate. This would include movie genres, release dates, actors, and even demographic data on users. The predictor we have demonstrated is only meant to show how this can be done using MapReduce and how the data clustered using MapReduce can be applied. Figure 4.3 shows a chart comparing the various predictors.

4.2 Movie Recommendations

This web application provides another demonstration of how the clustered data from the expensive clustering MapReduces (Step 4) can be used. This data has been loaded into a MySQL database that can be queried by the web application. The idea is that when a user enters the name of a movie the database is queried to provide the names of several similar movies. The database schema are provided in Appendix C. The actual procedure for querying the database is:

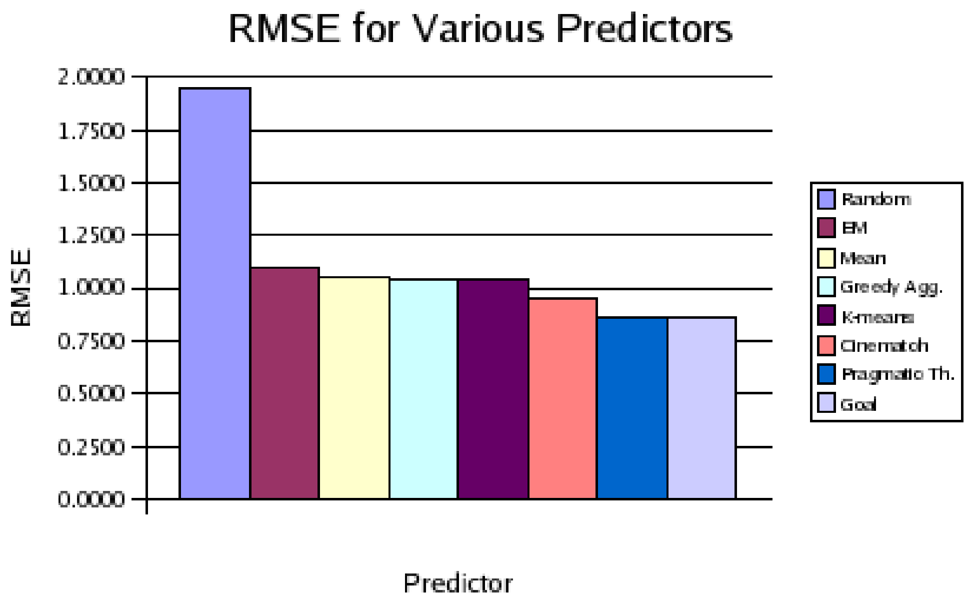

Figure 4.3. Predictor comparison chart

1. Check if the search movie title is a cluster center. If it is, then fetch several of the most similar movies to this cluster center as the result.

2. If the search movie title is not a cluster center:

   (a) Find the cluster center which it is closest to.

   (b) Fetch several movies most similar to this cluster center and return those as the result.

4.3 Hadoop and MySQL

The Predictor Preparation MapReduce presents an opportunity for an interesting collaboration between Hadoop and MySQL. This combination has interesting and exciting possibilities. Hadoop has the limitation that there is no practical way to search through a large amount of data that is needed during map operations. Database management systems in conjunction with Hadoop provide an alternative and additional method to provide input and output to MapReduce programs. MySQL is commonly deployed in environments requiring high availability and involving very large amounts of data. It can also provide various other benefits such as speed and scalability. Hence, MySQL is a feasible solution especially since it can be clustered on commodity hardware and can provide 99.999% availability. We were not able to run my experiments on a MySQL cluster, but that would be expected to provide additional benefits. MySQL cluster is designed to not have a single point of failure, and if properly designed any single node failing does not affect the cluster. MySQL cluster has 3 different types of nodes that interestingly parallels the node structure for Hadoop's MapReduce engine. The management node is used for configuration and monitoring of the cluster, and this similar to the jobtracker in Hadoop. Data nodes simply store the data as they do in Hadoop. A SQL node connects to data nodes and is able to perform information storage and retrieval from them. This node is similar to Hadoop task nodes. Generally it is preferable to have each node run on a separate host computer, as with Hadoop. Although it is feasible to have task/SQL nodes and data nodes run on the same hardware. Hadoop is different though in its setup in that the jobtracker is a single point of failure. If it dies during a job the job is lost and can not be resumed but must be restarted. Task nodes failing will not cause jobs to fail. Similarly SQL nodes failing will not disrupt the cluster. Hence, MySQL cluster seems to be most feasible if MySQL is paired with Hadoop in very large data set environments where high performance and high availability are required.

In the experimental setup we used a single MySQL database for testing. MySQL can handle a large number of queries and a large number of connections before experiencing noticeable performance loss. In the PredictorPrep MapReduce all the queries are made in the map step. Connections are opened in the `configure` method and closed in `close` method of the `Reduce` class. In the Hadoop configuration each node has a capacity of two map tasks at a time. Since one can not exactly control the total number of map tasks launched for a job, it is possible to control the number of maps running simultaneously for a job by increasing the number of tasks nodes used and measuring the performance. In this experiment we used a subset of the probe file and selected raters for 1,000 movies. This resulted in a total of over 67,000 predictions to be made. The PredictorPrep MapReduce queries the database between one to three times per movie to find several similar movies. Hence, for this particular experiment this resulted in 2,877 total queries, for which the running time for the various number of nodes are shown in Table 4.4. As shown in Table 4.4, as the number of maps running in parallel increases so does the running time. However, it is more than likely

TABLE 4.4 Hadoop and MySQL performance

Number of Nodes	Peak Parallel Maps	Running Time
1	2	20min 28sec
10	20	29min 25sec
20	40	29min 36sec
50	100	31min 1sec

that this is mostly due to the overhead on the Hadoop MapReduce engine of more maps to start and track than of the increased load on MySQL. A better test would be to use a fixed number of maps and vary the number of MySQL servers to determine the performance improvement with that approach. Also a larger number of queries in the hundred thousand to millions range would have to be used.

4.4 Insights on Hadoop

From our experiences during this study we feel that Hadoop should be considered as a means to a solution whenever very large amounts of data are involved. Hadoop should only be considered for time insensitive jobs that can or would be batch processed. Furthermore, from our experience Hadoop is ideal for solving problems that are easily partitioned into independent subtasks related by a particular key value and solveable using divide and conquer [22] techniques. It is difficult to determine whether Hadoop should be considered a Parallel or Distributed computing framework. Parallel computing is concerned with speeding up the execution speed of jobs by spreading the tasks involved amongst a group of nodes. Distributed computing on the other hand is more concerned with fault tolerance and reliability achieved by the added redundancy of back up machines for failover. Whilst Hadoop can process jobs faster than a single node, there are instances where a job will run much faster in Hadoop stand-alone mode than on a cluster of even just two machines. Furthermore, a programmer does not have any direct control over the execution speed of a job. In some cases especially for embarrassingly parallel [23] jobs a programmer may increase the number of computing nodes and also suggest a larger number of maps to be used. Hadoop's API allows a programmer to specify a value for the number of maps to be used. However, this is really only supplied as a hint to the system and in most cases is not the exact value of maps used in a particular Hadoop job. Hadoop employs some parallel computing concepts and paradigms such as mutual exclusion and data dependencies but this is transparent to the user. Hadoop also has some distributed computing characteristics such as checkpointing in case of failure and load balancing which are also transparent to the user.

4.4.1 How many maps and reduces?

An interesting issue when using MapReduce frameworks is determining how many maps and reduces to use for optimal performance. The documentation for Hadoop 0.18.1 indicates that the right number of reduces seems to be 0.95 or 1.75 (which is the reduce factor) multiplied by the number of task nodes multiplied by the maximum number of simultaneous reduces per task tracker. By default the maximum number of simultaneous reduces per task tracker is 2. Hence, using five task nodes the right number of reduces should either be 10 or 18. Using more reduce tasks lower the cost of failures but increases overhead on the framework and load balancing. Using 0.95 the reduces can all start immediately transferring map outputs as soon as individual maps complete. The method `setNumReduceTasks` in Hadoop 0.18.1 API `JobConf` class allows the exact number of reduce tasks to be specified. This is useful if one needs to know the number of output files for to be used as input in another MapReduce, but also can be useful if specifying the number of reduces is essential for ensuring correct results for some computation. An example of this is the Canopy Selection MapReduce where the number of reduces had to be set to one in order for it to work correctly.

Table 4.5 shows the effect of the number of reduces on the Data Preparation MapReduce (Step 1) and the canopy Selection MapReduce (Step 2) when using five task nodes. Figure 4.4 shows a chart of the time taken in seconds to complete each MapReduce against the reduce factor used. In most of the MapReduces we implemented the actual reduce step is rather cpu-light and often quite trivial. An example

TABLE 4.5 Effect of number of reduce tasks on performance

Factor	Num. Reduce Tasks	Data Preparation	Canopy Selection
0.1	1	14min 9sec	3min 18sec
0.50	5	12min 5sec	3min 21sec
0.95	**10**	**12min 10sec**	**4min 8sec**
1.50	15	12min 46sec	4min 10sec
1.75	**18**	**12min 56sec**	**4min 48sec**
2.50	25	13min 25sec	5min 30sec
5.0	50	14min 44sec	7min 43sec
10.0	100	18min 26sec	12min 20sec

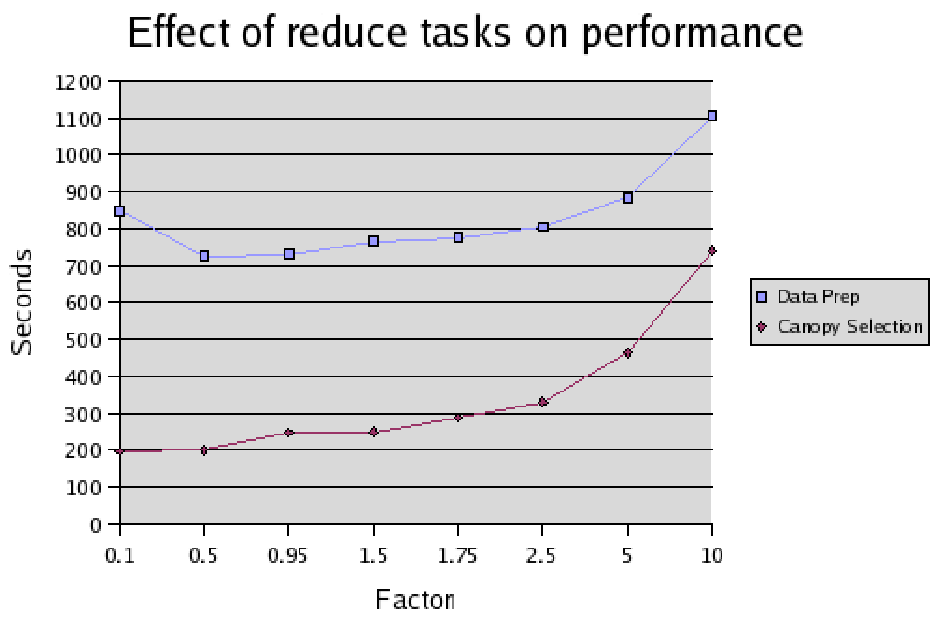

Figure 4.4. Plot of effect of reduce tasks on performance

of this is the Data Preparation MapReduce where the reduce step simply concatenates the raterID-rating pairs it receives for each movieID before outputting. However, the Canopy Selection MapReduce is an example of a non-trivial reduce step where actually the reduce steps perform the exact same operation as the map steps. In Figure 4.4 we see that for the trivial reduce Data Preparation optimal performance is achieved when the reduce factor is close to the recommended values. In this particular case 0.95 would have been a good value to use. However, for reduce factors outside of the recommended range the performance declines. For the Canopy Selection MapReduce we see that the pattern is similar where the best performance is achieved when the reduce factor is closer to the recommended values. However, in this case for low reduce factors the performance is still fairly good. Performance begins to decline more noticeably for large reduce factors. It is worth noting that the size of the data set used is only 2 GB and only five tasks nodes where used. In typical scenarios for which MapReduce was designed terabytes and data and hundreds tasks node would be involved. This would certainly change the performance results along with the complexity of the task the reduce step is required to perform. The number of maps tasks can not be exactly specified. The method `setNumMapTasks` of the `JobConf` class accepts an integer parameter for the number of maps but this is only a hint to the framework. From our experience the actual number of map tasks is never equal to the value specified but can be fairly close. The documentation states that 10-100 map tasks per task node is the best amount, although this can be increased to about 300 maps tasks for cpu-light jobs. In the Data Preparation MapReduce example (Step 1) there is about 2GB worth of input data contained in 17,770 text files. For this job we used 20 nodes. The framework automatically selected 17,770 maps resulting in almost 889 maps per node. Interestingly the average input file size for this job would be about 0.12MB. Hadoop was really designed for very large file sizes that could be split up into smaller chunks and distributed to task nodes. So it appears that if files are already small and do not need to be split up they are assigned to individual map tasks. Hence, it is not surprising that the number of maps is actually based on the number of input file chunks. Hadoop allows a file chunk size to be specified in the configuration property `mapred.min.split.size`, and by default is 64MB. Hence, with 2GB of data contained in fewer much larger files and a file chunk size set to 32MB, the result would be about 64 maps. Hence, only 1-6 tasks nodes would be required. The method `setNumMapTasks` could be used to set it higher if necessary for performance or design factors. Another factor to consider is that task setup involves some overhead so it is best if each map task takes at least one minute to run.

TABLE 4.6 Effect of number of map tasks on performance

Maps Rqstd	Incr. Factor	Actual Maps	Variance.	Maps/Node	Running Time
84	**1**	**84**	**0%**	**16.8**	**12min 48sec**
126	1.5	131	+3.97%	26.2	13min 3sec
420	5.0	427	+1.67%	85.4	17min 21sec
840	10.0	847	+0.83%	169.4	23min 49sec
4200	50.0	4206	+0.14%	841.2	1hr 15min 59sec

In Table 4.6 the first row in bold print shows the default number of maps automatically assigned by Hadoop. Taking this value as the default, subsequent jobs are run with increased maps by calling `setNumMapTasks` with the appropriate value. As shown in Table 4.6 the actual number of maps launched by Hadoop is not the same and the percentage variance is indicated. As shown the variance decreases as the number of maps increases. This experiment does show that an inappropriate number of maps can quickly degrade performance much faster than a poor choice of the number of reduces. Figure 4.5 clearly shows this performance loss.

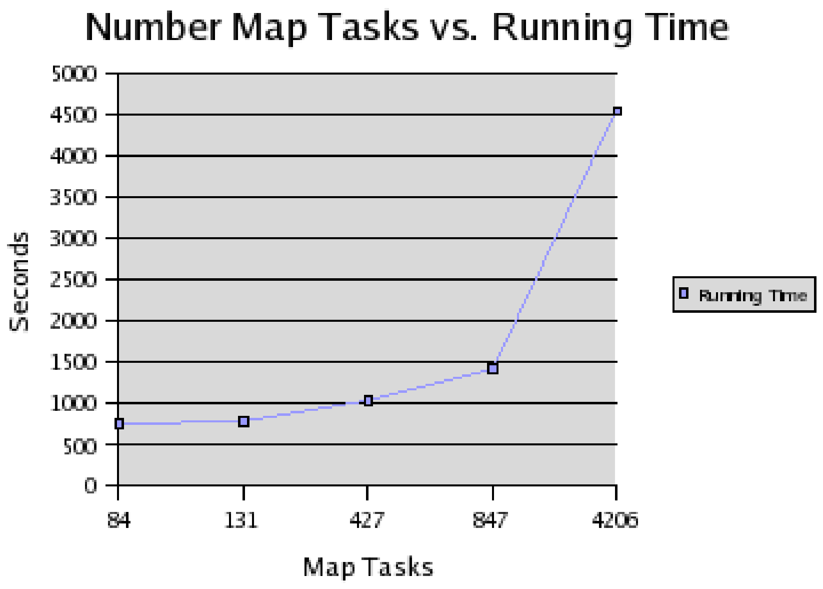

Figure 4.5. Plot of effect of map tasks on performance

Chapter 5

CONCLUSIONS

5.1 What have we done so far?

We have demonstrated how MapReduce can be used to implement various data clustering algorithms. We have also shown how this clustered data may be used to provide movie predictions and recommendations for the Netflix problem. In the process we also demonstrated how MapReduce frameworks can collaborate with Database Management Systems allowing for interesting possibilities.

5.2 Future directions

The direction of continuing future research will be focused on improving the RMSE of the Netflix movie predictions. Although this was never the intent of the study, the work achieved thus far presents a unique opportunity to address a problem that has gained considerable worldwide attention. In this report the predictor presented is just meant to show how this can be done using MapReduce. Obviously a predictor can not just be based on movieIDs, raterIDs and scores. Other factors such as movie release dates, genres, actors, studios, and so forth could be utilized to produce better results. However, the data clustering involved is still an important step in the process. In future work different distance metrics could also be explored such as Pearson correlation coefficient [21], which may be appropriate for this problem.

Another interesting topic would be to explore HBase in comparison to more common database management systems. HBase is a distributed database that runs on Hadoop Distributed File System and can serve as input and output for MapReduce jobs. Especially where very large amounts of data are involved. It would be interesting to see how HBase compares to other database management systems such as Oracle, PostgreSQL, and MySQL. Particularly in terms of performance, availability, and reliability

5.3 Conclusion

MapReduce is a feasible solution to processing problems involving large amounts of data. Especially for problems that can easily be partitioned into independent sub tasks that can be solved in parallel. Hadoop is an open source MapReduce implementation featured in this study. Hadoop should be considered only for non-time sensitive tasks that can be batch processed. An example is certain types of data clustering, such as the Netflix data clustering presented in this report. As demonstrated the most challenging part with designing MapReduce programs is usually determining inputs, outputs and expressing a problem in MapReduce terms. In the future it is feasible that the demand for cluster computing will grow as problem sets become larger and more interest develops in the field. MapReduce will undoubtedly become a more popular solution and much used paradigm.

REFERENCES

[1] A.K. Jain, M.N. Murty, and P.J. Flynn. *Data Clustering: A Review.* ACM, New York, NY, USA, 1999.

[2] Tom White, *Hadoop: The definitive Guide.* O'Reilly Media, Sebastopol, California, USA, 2008.

[3] Jeffrey Dean, and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation.*, pp. 137-150, USENIX Association, Berkley, CA, USA, 2004.

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System." *Proceedings of the nineteenth ACM symposium on Operating systems principles.* , pp. 29-43, ACM, New York, NY, USA, 2003.

[5] Netflix. *http://www.netflix.com*

[6] Netflix Prize. *http://www.netflixprize.com*

[7] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. "Using collaborative filtering to weave an information tapestry" *Communications of the ACM.* 35 (12): 6170, ACM, New York, NY, USA, 1992.

[8] H. Steinhaus, *Sur la division des corp materiels en parties*, Bulletin L'Academie Polonaise des Science, Paris, France, 1956.

[9] E. M. Voorhees, *Implementing agglomerative hierarchical clustering algorithms for use in document retrieval*, Pergamon Press, Tarrytown, NY, USA, 1986.

[10] A. P. Dempster, N. M. Laird and D. B. Rubin, *Maximum Likelihood from Incomplete Data via the EM Algorithm*, Blackwell Publishing, Oxford, England, UK, 1977.

[11] A. McCallum, K. Nigam, and L. H. Ungar. *Efficient Clustering of High Dimensional Data Sets with Application to Reference Matching.* Association for Computing Machinery (ACM), New York, NY, USA, 2000.

[12] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. *An Efficient k-Means Clustering Algorithm: Analysis and Implementation.* IEEE Computer Society, Washington, DC, USA, 2002

[13] D. Arthur and S. Vassilvitskii. *K-means++: The Advantage of Careful Seeding.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2007.

[14] M.W. Mak, S.Y. Kung, and S.H. Lin. *Biometric Authentication: A Machine Learning Approach.* Prentice Hall Professional, Upper Saddle River, New Jersey, USA, 2005.

[15] I. D. Dinov. *Expectation Maximization and Mixture Modeling Tutorial.* University of California, Los Angeles, USA, 2008

[16] P.C. Mahalanobis. *On the generalised distance in statistics.* Proceedings National Institute of Science, Calcutta, India, 1936.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data." *ACM Transactions on Computer Systems*, vol. 26(4), ACM, New York, NY, USA, 2008.

[18] MySQL. *http://www.mysql.com*

[19] R. De Maesschalck, D. Jouan-Rimbaud, and D.L. Massart *The Mahalanobis distance.* Texas A&M University, College Station, TX, USA, 2000.

[20] M. H. DeGroot, and Mark J. Schervish. *Probability and Statistics.* Addison-Wesley, London, England, UK, 2002.

[21] J. L. Rodgers and W. A. Nicewander. "Thirteen ways to look at the correlation coefficient." *The American Statistician.* 42(1):5966, 1988.

[22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein *Introduction to algorithms.*, MIT Press, Cambridge, Massachusetts, USA, 2001.

[23] I. Foster *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison Wesley, London, England, UK, 1995.

Appendix A

EXPERIMENTAL SETUP

- All experiments were conducted using Hadoop 0.18.1 and Java (J2SE) version 1.6 at Boise State University College of Engineering's Onyx and Beowulf computing clusters.
- Onyx is a 32 node Gigabit Ethernet Beowulf cluster. Nodes are Intel Pentium 4 2.8 GHz processors with 1 GB RAM running Red Hat Fedora 7.
- Beowulf is a 50 node + Gigabit Ethernet Beowulf cluster. Nodes are mostly dual core Intel Xeon 2.4GHz processors with 1 GB RAM (some 2 GB and 4 GB available) running CentOS release 5.2.
- MySQL 5.0.45 is the database used running on a dual core 2.4 GHz Intel Xeon node with 4 GB RAM.
- Netflix web application demo is temporarily hosted at:

  `http://onyx.boisestate.edu/~mngazimb/NetflixWeb/`

  This application requires PHP 5+ and has been designed for Mozilla Firefox 1.5+ browsers. Javascript must be enabled.

Appendix B

UML DIAGRAMS

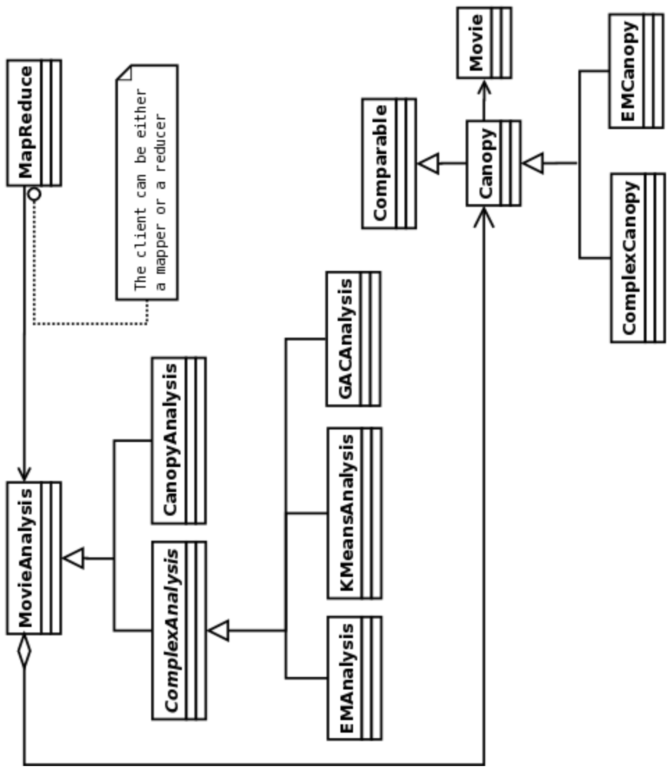

Figure B.1. Netflix MapReduce class diagram

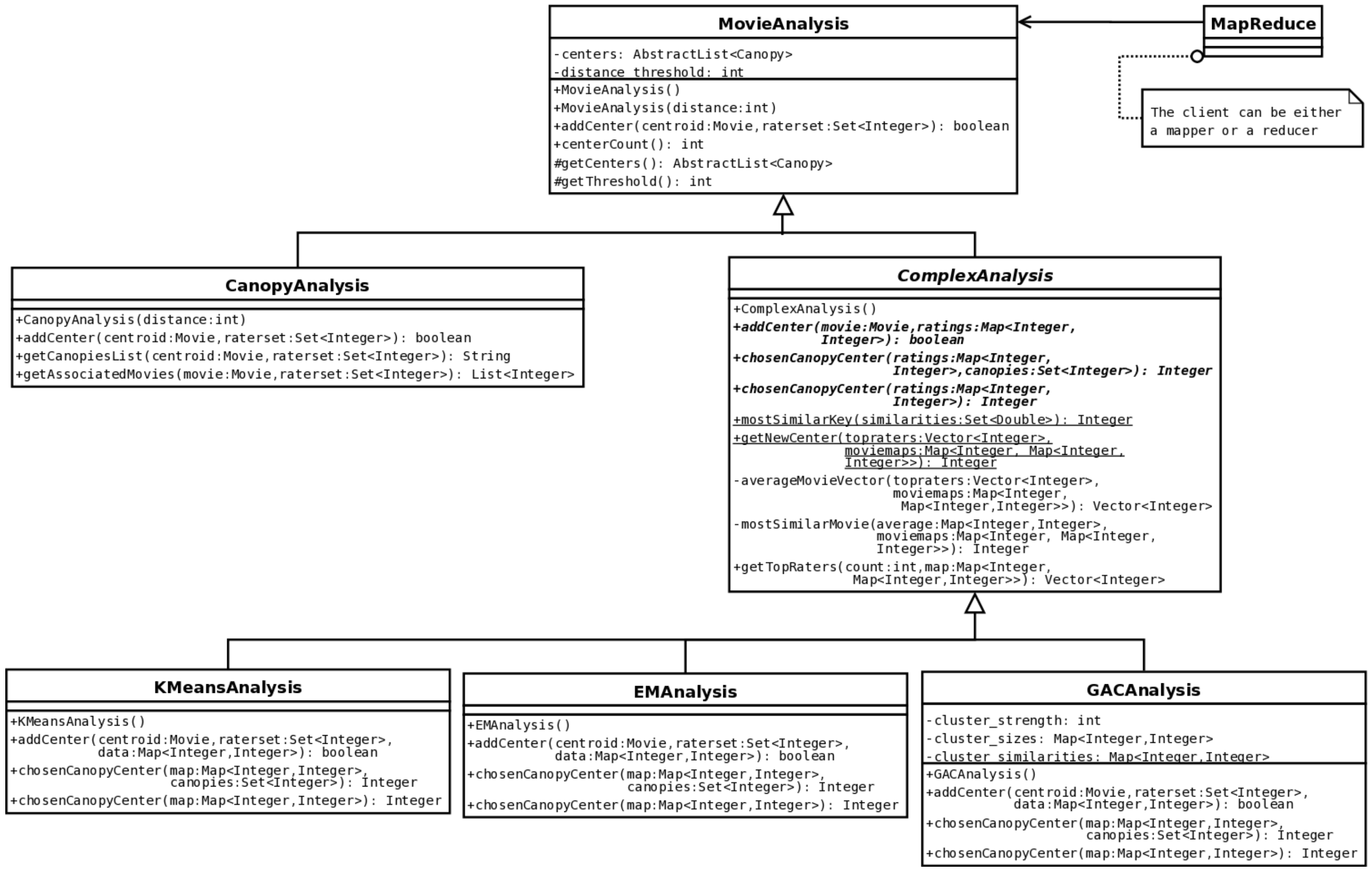
MapReduce
The client can be either a mapper or a reducer
MovieAnalysis
-centers: AbstractList<Canopy>
-distance_threshold: int
+MovieAnalysis()
+MovieAnalysis(distance:int)
+addCenter(centroid:Movie,raterset:Set<Integer>): boolean
+centerCount(): int
#getCenters(): AbstractList<Canopy>
#getThreshold(): int
CanopyAnalysis
+CanopyAnalysis(distance:int)
+addCenter(centroid:Movie,raterset:Set<Integer>): boolean
+getCanopiesList(centroid:Movie,raterset:Set<Integer>): String
+getAssociatedMovies(movie:Movie,raterset:Set<Integer>): List<Integer>
ComplexAnalysis
+ComplexAnalysis()
+addCenter(movie:Movie,ratings:Map<Integer, Integer>): boolean
+chosenCanopyCenter(ratings:Map<Integer, Integer>,canopies:Set<Integer>): Integer
+chosenCanopyCenter(ratings:Map<Integer, Integer>): Integer
+mostSimilarKey(similarities:Set<Double>): Integer
+getNewCenter(topraters:Vector<Integer>, moviemaps:Map<Integer, Map<Integer, Integer>>): Integer
-averageMovieVector(topraters:Vector<Integer>, moviemaps:Map<Integer, Map<Integer,Integer>>): Vector<Integer>
-mostSimilarMovie(average:Map<Integer,Integer>, moviemaps:Map<Integer, Map<Integer, Integer>>): Integer
+getTopRaters(count:int,map:Map<Integer, Map<Integer,Integer>>): Vector<Integer>
KMeansAnalysis
+KMeansAnalysis()
+addCenter(centroid:Movie,raterset:Set<Integer>, data:Map<Integer,Integer>): boolean
+chosenCanopyCenter(map:Map<Integer,Integer>, canopies:Set<Integer>): Integer
+chosenCanopyCenter(map:Map<Integer,Integer>): Integer
EMAnalysis
+EMAnalysis()
+addCenter(centroid:Movie,raterset:Set<Integer>, data:Map<Integer,Integer>): boolean
+chosenCanopyCenter(map:Map<Integer,Integer>, canopies:Set<Integer>): Integer
+chosenCanopyCenter(map:Map<Integer,Integer>): Integer
GACAnalysis
-cluster_strength: int
-cluster_sizes: Map<Integer,Integer>
-cluster_similarities: Map<Integer,Integer>
+GACAnalysis()
+addCenter(centroid:Movie,raterset:Set<Integer>, data:Map<Integer,Integer>): boolean
+chosenCanopyCenter(map:Map<Integer,Integer>, canopies:Set<Integer>): Integer
+chosenCanopyCenter(map:Map<Integer,Integer>): Integer

Figure B.2. MovieAnalysis hierarchy UML

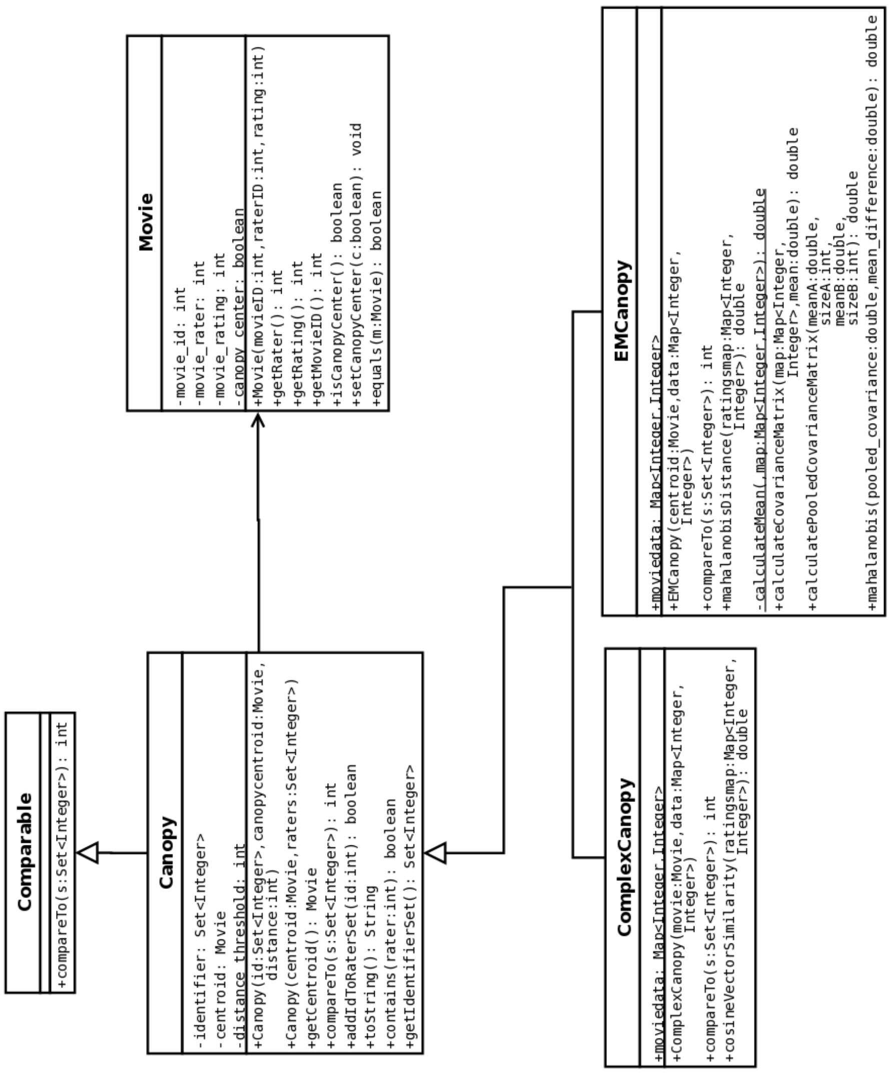
Comparable
+compareTo(s:Set<Integer>): int
Canopy
-identifier: Set<Integer>
-centroid: Movie
-distance_threshold: int
+Canopy(id:Set<Integer>,canopycentroid:Movie, distance:int)
+Canopy(centroid:Movie,raters:Set<Integer>)
+getCentroid(): Movie
+compareTo(s:Set<Integer>): int
+addIdToRaterSet(id:int): boolean
+toString(): String
+contains(rater:int): boolean
+getIdentifierSet(): Set<Integer>
Movie
-movie_id: int
-movie_rater: int
-movie_rating: int
-canopy_center: boolean
+Movie(movieID:int,raterID:int,rating:int)
+getRater(): int
+getRating(): int
+getMovieID(): int
+isCanopyCenter(): boolean
+setCanopyCenter(c:boolean): void
+equals(m:Movie): boolean
ComplexCanopy
+moviedata: Map<Integer,Integer>
+ComplexCanopy(movie:Movie,data:Map<Integer, Integer>)
+compareTo(s:Set<Integer>): int
+cosineVectorSimilarity(ratingsmap:Map<Integer, Integer>): double
EMCanopy
+moviedata: Map<Integer,Integer>
+EMCanopy(centroid:Movie,data:Map<Integer, Integer>)
+compareTo(s:Set<Integer>): int
+mahalanobisDistance(ratingsmap:Map<Integer, Integer>): double
-calculateMean(map:Map<Integer,Integer>): double
+calculateCovarianceMatrix(map:Map<Integer, Integer>,mean:double): double
+calculatePooledCovarianceMatrix(meanA:double, sizeA:int, meanB:double, sizeB:int): double
+mahalanobis(pooled_covariance:double,mean_difference:double): double

Figure B.3. Canopy hierarchy UML

Appendix C

DATABASE SCHEMA

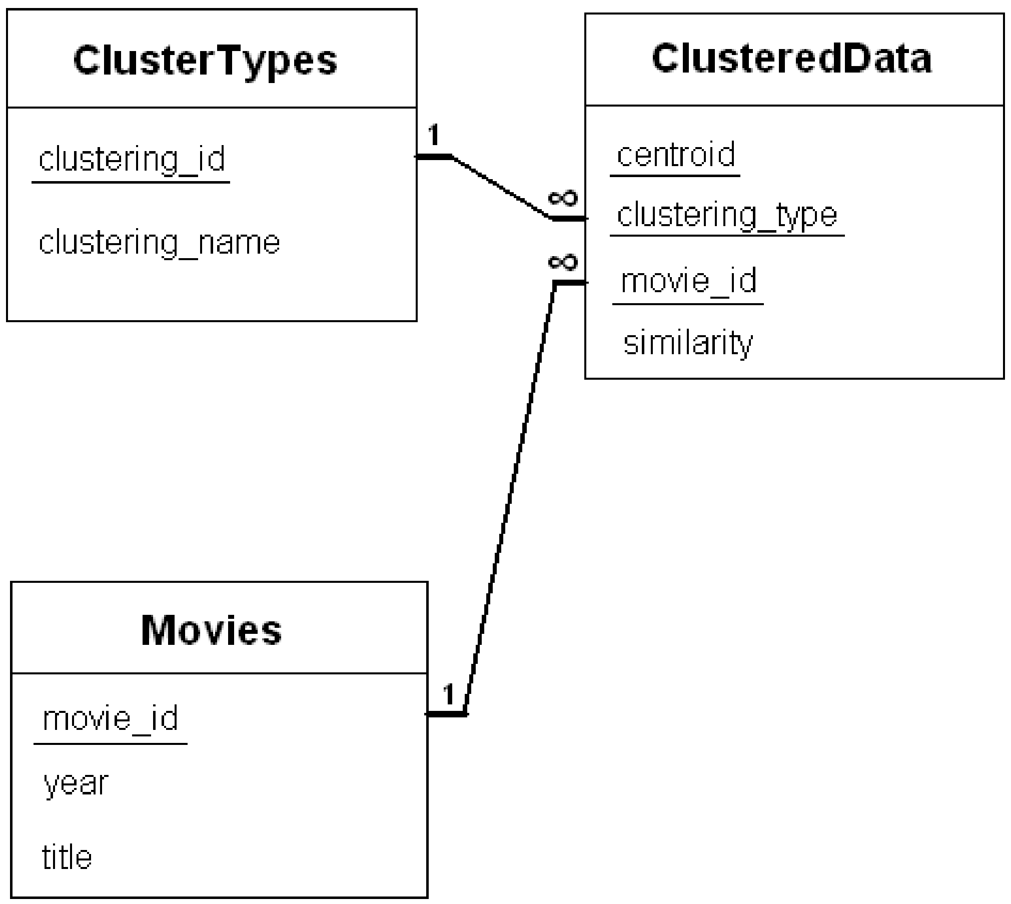

Figure C.1. Netflix database schema