Calculating the Jaccard Similarity Coefficient with Map Reduce for Entity Pairs in Wikipedia

Jacob Bank (jeb369), Benjamin Cole (bsc36)
Wikipedia Similarity Team

December 16, 2008

Contents

0.1 Abstract

The goal of the project was to create a tool to analyze the huge amounts of data associated with large-scale social networks on the web. Specifically, the project was to create a Map Reduce program to calculate the Jaccard similarity coefficient between users of Wikipedia based on co-occurrence of page edits. The program was then generalized to calculate the Jaccard similarity between entities in any arbitrary column of a data set, based upon co-occurrence with another arbitrary column.

The program was implemented in Java with the MapReduce programming technique. Using the Hadoop open-source framework, the jobs were run on the approximately 50-node cluster at the Cornell Center for Advanced Computing.

In the writing and testing of the program, a few interesting findings were discovered about both MapReduce programming and the underlying data set of Wikipedia page edits. First, parallelizing computationally intensive tasks was critical to performance since some reduce tasks performed computations on extremely long lists. Doing many separate linear time computations proved to be much faster than doing quadratic computations on these long lists (in some cases reducing running time from days to minutes). In running the jobs, the program took significantly longer to calculate similarity between pages than similarity between users. This difference in run time seemed to arise due to a few users with an extremely high number of page edits. Computations on these very long lists of pages dominated the running time and made page similarity calculations much slower. This interesting asymmetry in the data further showed that MapReduce jobs can often be dominated in running time by a few extremely long lists in reduce tasks.

0.2 Introduction

The purpose of the Web Laboratory is to provide data and computing tools for research about the Web and information on the Web. One of the teams working towards this goal is the Wikipedia Similarity Group. The team has been working with authorship data from Wikipedia to compute the Jaccard similarity coefficient [1] for pairs of users and pairs of pages. This document describes the algorithm used to compute the coefficient, its implementation, and obstacles overcome along the way.

Due to the large-scale nature of the data and the computationally intensive task at hand, the team decided to tackle this problem using the Map Reduce programming paradigm running on the Web Lab's Hadoop cluster. Hadoop [2] is an open source framework that provides both reliability and data motion for applications. It implements the Map Reduce [3] model, in which an application's work load is divided into many smaller segments and distributed over the nodes of the cluster. Hadoop also provides a reliable distributed file system (HDFS) that stores the data across the nodes. This in turn allows for extremely high aggregate bandwidth over the entire cluster.

0.3 The Jaccard Similarity Coefficient

Also known as the Jaccard index, the Jaccard similarity coefficient is a statistical measure of similarity between sample sets. For two sets, it is defined as the cardinality of their intersection divided by the cardinality of their union. Mathematically,

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

In the case of the Wikipedia data, a slightly different approach is required. For a pair of entities, the calculation is performed on sets of entities with which the elements of the pair occur. For example, in calculating the similarity of pairs of pages, the numerator is the number of users that have edited both pages, and the denominator is the number of users who have edited either or both. Mathematically,

$$J(A, B) = \frac{|X \cap Y|}{|X \cup Y|}$$

where X and Y correspond to the sets of entities that occur with A and B, respectively.

0.4 The Algorithm

To compute the Jaccard similarity coefficient for a pair of X elements, calculation of two quantities is needed. The first is the number of Y elements that occur with both X elements, and the second is the number of Y elements that occur with one or both. Calculating the first quantity was relatively straightforward, but for the second quantity, one must first calculate the number of Y elements that occur with the first X element plus the number of Y elements that occur with the second X element and then subtract the number of Y elements that occur with both. Finally, the first quantity is divided by the second quantity to arrive at the proper coefficient.

0.4.1 *Pass One*

XY Identity Map

This map method simply takes in a key/value pair and returns them unchanged.

Input: $\underline{Text}$, $Text$
Output: $\underline{Text}$, $Text$

$\underline{X}, Y \rightarrow \underline{X}, Y$

Count Y Reduce

This reduce method takes a key and the set of all corresponding values, and computes the cardinality of the set of values, C, and returns the input key with each input value and C. This quantity, C, is eventually used the compute the denominator of the Jaccard similarity coefficient.

Input: <u>Text</u>, {Text}
Output: <u>Text</u>, CPair

$\underline{X}, \{Y\} \rightarrow \underline{X}, (Y, C) \quad where\ C = |\{input\ values\}|$

0.4.2 *Pass Two*

XY to YX Map

This map method takes the key/value pair and switches the *String* in the key with the *String* in the value. It then returns the modified key and value. Using the Y element as the key allows the combine stage to collect all of the X elements that correspond to each Y element. This enables the later collection of all pairs of X that occur with a single Y.

Input: <u>Text</u>, CPair
Output: <u>Text</u>, CPair

$\underline{X}, (Y, C) \rightarrow \underline{Y}, (X, C)$

Split Reduce

This reduce task takes in a key and set of values, splits up the set of values, and outputs an empty key with each of the smaller lists of values. This is done to load balance the collection of pairs in the next map method. Since the logic involved with this step is fairly complex, the code is included in the appendix.

Input: <u>Text</u>, {CPair}
Output: <u>Text</u>, {CPair}

$\underline{Y}, \{(X, C)\} \rightarrow \underline{R}, \{(X, C)\}$ *where R is an empty placeholder*

0.4.3 *Pass Three*

Pair Collect Map

This map takes in the empty Text key and the list of values. It then creates a series of X_i, X_j pairs with the first element in the value list and each of the following elements. A list of size n will produce n-1 such pairs, one for each element in the list, except for the first.

Input: <u>Text</u>, {CPair}
Output: <u>Pair</u>, IntWritable

$\underline{R}, \{(X, C)\} \rightarrow \underline{(X_i, X_j)}, C_{ij}$ *where* $C_{ij} = C_i + C_j$

Normalization Reduce

This reduce method calculates the number of co-occurences of X_i and X_j by counting the number of elements in the set of input values. It then divides the number of co-occurences by the total number of Y elements that occur with one or both of X_i and X_j.

Input: <u>Pair</u>, {IntWritable}
Output: <u>Pair</u>, FloatWritable

$$\underline{(X_i, X_j)}, \{C_{ij}\} \rightarrow \underline{(X_i, X_j)}, \frac{|\{C_{ij}\}|}{C_{ij} - |\{C_{ij}\}|}$$

0.5 Experiments and Results

0.5.1 *Varying Input Size*

The main purpose of the MapReduce programming technique is to handle extremely large data sets, so testing how well the program scaled up was extremely important. This experiment involved testing the program on data slices of different sizes, ranging from ten lines to one million lines of page edit data. The program calculated the similarities between pages and between users on data sets of 10, 100, 1,000, 10,000, 100,000, and 1,000,000 lines, where each line contains the information about a single page edit. Each trial was performed three times and the running times were averaged.

The graph showing the results can be seen in Figure 1 on the next page. This experiment produced a few interesting results. First, it indicated that there was a fixed overhead of time for running Hadoop jobs on the cluster. All the jobs for calculating user similarity, and all jobs for page similarity except for the largest input size, took a little over one minute to run. Even on a trivial data set of ten lines, the job ran no faster than one minute. This seemed to show that the overhead for running a Hadoop job with three passes was around one minute. The experiment also achieved its main goal, which was to show the scalability of the program. The user similarity calculation operated within the Hadoop overhead on all trials (data sets of up to 1,000,000 lines). The page similarity calculation operated within the Hadoop overhead on all trials with data sets up to 100,000 lines. These results demonstrated that the ultimate goal of the project, scalability to large amounts of data, was achieved. The final interesting result was the asymmetry between user similarity and page similarity calculations. On the 1,000,000 line data set, the page similarity calculation took about three and a half minutes, as opposed to one minute for the user similarity calculation.

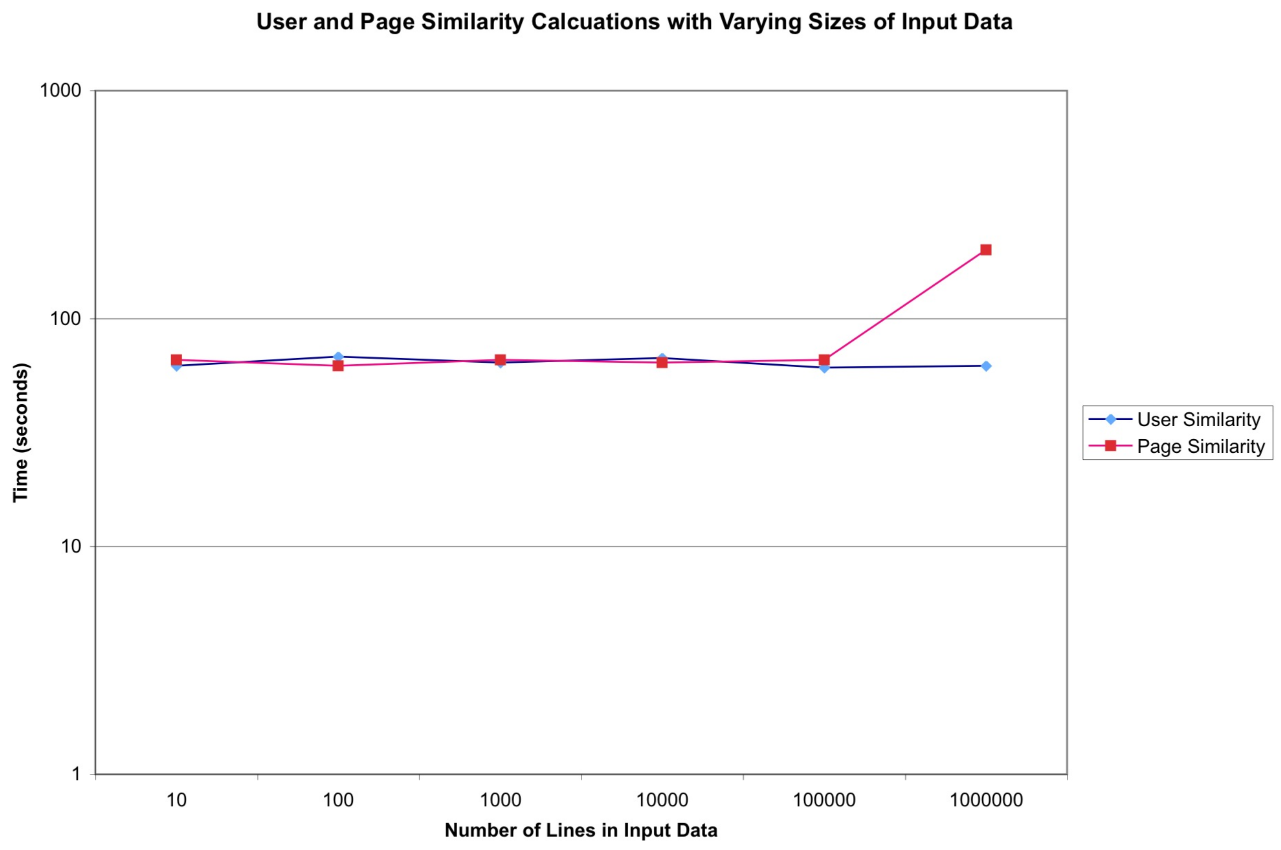

Figure 1: Results for the experiments of varying input size on similarity calculations for users and pages.

0.6 Key Insights

0.6.1 *Parallelization of Computationally Intensive Tasks*

The elegance of the algorithm lies in the fact that, at every step, each map and reduce is only $O(n)$. A naïve and inefficient initial approach to collecting all pairs in a list of elements was to use a selection-sort-like algorithm of nested loops. For each element in the list, this algorithm created a pair with it and every entity following it in the list. However, this was an $O(n^2)$ operation, and proved to be an unacceptable bottleneck in the computations.

Since the size of input with this type of data often follows a power law distribution, there is generally a major disparity in the size of sets combined for reduce tasks. Thus, while an $O(n^2)$ operation would generally be acceptable on input of moderate sizes (on lists up to 1000 elements), this data set produced a small number of inputs much larger in size (lists with 50,000 and above elements). Therefore, to be able to process these long lists, the pair collection operation had to be broken down into two operations of $O(n)$. Based on this work, it seems that sacrificing the benefit of fewer passes over the data in order to parallelize an otherwise computationally intensive task greatly increases overall efficiency.

0.6.2 *Asymmetry between User and Page Data*

An interesting observation discovered by running the algorithm on the Wikipedia data was an extreme asymmetry between the time needed to compute the similarity between pairs of pages and that needed for pairs of users. Specifically, the time necessary to compute the similarity between pairs of pages was higher than the time necessary for pairs of users on large datasets. Upon examining the data, we found that the sets of pages that users have edited tended to be much larger than the sets of users who have co-edited pages. In a 600,000 line data set, the longest list associated with a user was about 50,000 as opposed to about 2,000 for the longest list associated with a page. Although initially surprising, this finding is actually intuitive given an understanding of the data and the Wikipedia community. While a page is usually edited by a small, finite number of users, some users, usually bots, have actually edited a significant portion of all the pages on Wikipedia.

0.6.3 *Lessons about Hadoop*

The major learnings about Hadoop revolved around the way Hadoop manages memory, and further, the way a programmer using Hadoop should handle memory. The first major obstacle involved the reduce method's input value iterator. The initial naïve approach assumed that the iterator would return a new object every time the $next()$ method was called. To the contrary though, Hadoop actually uses the same object repeatedly by simply replacing the values. While this approach would not make much sense to programmers unaccustomed to dealing with data analysis at scale, it is in fact justified. By using the same object, Hadoop reduces the amount of memory used by a factor of n.

In turn, "Cannot Allocate Memory" errors appeared during the pair collect operation, showing the strategy of creating a new object for every pair in memory was not scalable. Thus, the new implementation employed the same approach, creating and reusing a single object while simply replacing the values after each output collection. In this way, memory usage was reduced by a factor of n.

0.7 Conclusions and Future Work

This work represents only a small first step in using the MapReduce programming technique in the analysis of large social networks. There are a number of direct extensions we can see making to this particular work to make it more useful in real world analysis. First, we could add the ability to filter users and pages by certain characteristics. Another potential direction would be to filter the data into time slices to track trends over time. Furthermore, we do not currently weight a co-occurence that appears many times over a co-occurence that appears only once. All of these would be useful potential expansions on our current work.

In addition to adding to this project, there are a number of related tasks that we would like to do using MapReduce programming, such as calculating network metrics. Specifically, we could write programs that could find clustering coefficients and degree distribution of networks.

An interesting application of these tools would be to compare the development of different language versions of Wikipedia. Do their networks follow the same patterns as the early versions of the English version of the site? Or do the patterns of new language versions quickly match the current English version? Once we have a broader set of tools, we can imagine tackling many other interesting questions about online social networks.

0.8 Acknowledgements

We would like to thank our advisors Professor William Arms and Professor Daniel Cosley as well as their graduate students Asif-ul Haque and Vladimir Barash for their guidance. We would also like to thnk Lucia Walle of the Cornell Center for Advanced Computing for her support in maintaining the Hadoop cluster.

This work is funded in part by National Science Foundation grants CNS-0403340, DUE-0127308, SES-0537606, IIS-0634677, and IIS-0705774. The Web Lab is an NSF Next Generation Cyberinfrastructure project.

Bibliography

[1] Paul Jaccard. Etude comparative de la distribution florale dans une portion des Alpes et des Jura. In *Bulletin del la Socit Vaudoise des Sciences Naturelles*, volume 37, pages 547-579.

[2] Hadoop. http://hadoop.apache.org/

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 137-149, 2004.

0.9 Appendix: Source Code

0.9.1 MapRed.java

```
import java.io.*;
import java.util.*;

//Hadoop imports
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

/**
 * Map-Reduce job to find similarity between x's
 * i.e. similarity(xi,xj) = (number of y's they have in common)/(sum of y's associated with xi and xj)
 * @author Ben Cole and Jacob Bank
 */
public class MapRed extends Configured implements Tool{

/**
 * Map class that maps a text input of x y to a text output of x y
 */
public static class XYIdentityMap extends MapReduceBase implements Mapper<LongWritable, Text,
Text, Text>{
/**
* @param ikey Dummy parameter required by Hadoop for the map operation, but it does nothing.
* @param ival Text used to read in necessary data, in this case each line of the text will be in the format
of ( x y )
* @param output OutputCollector that collects the output of the map operation, in this case a (Text, Text)
pair representing the (x,y)
 * @param reporter Used by Hadoop, but we do not use it
 */
public void map(LongWritable ikey, Text ival, OutputCollector<Text, Text> output, Reporter reporter)
throws IOException{
                        StringTokenizer st= new StringTokenizer(ival.toString());
                        Text okey= new Text(st.nextToken());
                        Text oval= new Text(st.nextToken());
                        output.collect(okey, oval);
                }
}
/**
 * Reduce class that counts all of the y values for each x value
 */
public static class CountYReduce extends MapReduceBase implements Reducer<Text, Text, Text,
CPair>{
/**
 * @param ikey The x we are performing the y count on
 * @param vlist The list of y values associated with x
 * @param output OutputCollector that collects a (Text, CPair) pair, where the Text is the x and the CPair
  is a (y,count) pair
  * with count being the total number of y values associated with x
  * @param reporter Used by Hadoop, but we do not use it
  */
public void reduce(Text ikey, Iterator<Text> vlist, OutputCollector<Text, CPair> output, Reporter reporter)
throws IOException{
                        // Traverse the <key, vlist>
                    Set<Text> s = new TreeSet<Text>();
                        while(vlist.hasNext()){
                                Text t = vlist.next(); // returns the same object with a different value each time
                                s.add(new Text(t));
                        }
                        for (Text t : s) {
                            output.collect (ikey, new CPair(t.toString(), s.size()));
                        }
                }
}
/**
```

```
 * Map class that takes in data of the form (x y count) and outputs data with y as the key and a CPair of
 (x,count) as the value
 */
public static class XYtoYXMap extends MapReduceBase implements Mapper<LongWritable, Text, Text,
CPair>{
/**
 * @param ikey Dummy parameter required by Hadoop for the map operation, but it does nothing.
 * @param ival Text used to read in necessary data, in this case each line of the text will be in the format
 of ( x y count)
 * @param output OutputCollector that collects the output of the map operation, in this case a (Text,
 CPair) pair representing (y, (x,count))
 * @param reporter Used by Hadoop, but we do not use it
 */
public void map(LongWritable ikey, Text ival, OutputCollector<Text, CPair> output, Reporter reporter)
throws IOException{
                        StringTokenizer st= new StringTokenizer(ival.toString());
                        String x = st.nextToken();
                        String y = st.nextToken();
                        String count = st.nextToken();
                        Text okey= new Text(y);
                        CPair oval= new CPair(x, Integer.parseInt(count));
                        output.collect(okey, oval);
                }
}
/**
 * Reduce class that splits a list of size n into many smaller lists of sizes 1 to chunk size + 1
 * This is done to further parallelize the pair collection process, which was a bottleneck in earlier
 versions due to the
 * computational difficulty of performing an n-squared operation on a very long list, and also to balance the load by
 * ensuring that no list is larger than the chunk size.
 * Now, it will be many seperate order of n operations.
 */
public static class SplitReduce extends MapReduceBase implements Reducer<Text, CPair, Text,  Text>{
/**
 * @param ikey Dummy parameter required by Hadoop.
 * @param vlist The list of (x,count) pairs associated with each y
 * @param output OutputCollector that collects a String representing a list of (x,count) pairs
 * @param reporter Used by Hadoop, but we do not use it
 */
public void reduce(Text ikey, Iterator<CPair> vlist, OutputCollector<Text, Text> output, Reporter reporter)
throws IOException{
                    Vector<String> chunks = new Vector<String>();
                    final int c = 500;
                    int counter = 0;
                    Text key = new Text();
                    Text value = new Text();
                    while(vlist.hasNext())
                    {
                        String current = vlist.next().toString();
                        for(String chunk: chunks){
                            // The key is modified to avoid collisions in the next reduce.
                            key.set("");
                            value.set(current + " " + chunk);
                            output.collect(key, value);
                        }
                        if(counter % c == 0) chunks.add(current);
                        else chunks.set(chunks.size()-1, current + " " + chunks.get(chunks.size()-1));
                        counter++;
                    }
            }
}
/**
 * Map to collect pairs
 */
public static class PairCollectMap extends MapReduceBase implements Mapper<LongWritable, Text,
Text, IntWritable>{
/**
 * @param ikey Dummy parameter required by Hadoop for the map operation, but it does nothing.
 * @param ival Text used to read in necessary data, in this case each line of the text will be a list of
 CPairs
 * @param output OutputCollector that collects the output of the map operation, in this case a (Text, IntWritable) pair representing
 * @param reporter Used by Hadoop, but we do not use it
 */
public void map(LongWritable ikey, Text ival, OutputCollector<Text, IntWritable> output, Reporter
reporter) throws IOException{
                    String list = ival.toString();
                    ArrayList<CPair> al = new ArrayList<CPair>();
                    Scanner sc = new Scanner(list);

                    // Converts the text string to CPairs
                    while(sc.hasNext()){
                        String x = sc.next();
                        if(sc.hasNext()){
                            int c = sc.nextInt();
                            al.add(new CPair(x,c));
                        }
                    }
                    // Iterate over the CPairs to create Text objects
                    Text t = new Text();
                     for (int i = 1; i < al.size(); i++){
                        String x = al.get(0).getX();
                        String y = al.get(i).getX();
                        if(x.compareTo(y) > 0) t.set(x + "      " + y);
                        else t.set(y + "        " + x);
                        output.collect(t, new IntWritable(al.get(0).getCount() + al.get(i).getCount()));
                    }
                }
 }
/**
 * Reduce class that does the final normalization.
 * Takes in data of the form (xi xj (sum of counts))
 * Outputs the similarity between xi and xj as (the number of y values xi and xj are associated with)/(the
 sum of the counts)
 */
public static class NormalizationReduce extends MapReduceBase implements Reducer<Pair,
IntWritable, Text, FloatWritable>{
/**
 * @param ikey The xi xj pair for which we are calculating the similarity
 * @param vlist The list of ((xi,xj),sum of counts) pairs
 * @param output OutputCollector that collects a (Text, FloatWritable) pair. where the Text is the (xi,xj)
 pair and the FloatWritable is their similarity
 * @param reporter Used by Hadoop, but we do not use it
 */
public void reduce(Text ikey, Iterator<IntWritable> vlist, OutputCollector<Text, FloatWritable> output,
Reporter reporter) throws IOException{

                        float count = 0;
                        float sumcounts = 0;
                        while (vlist.hasNext()){
                                sumcounts = (float)vlist.next().get();
                                count++;
                        }
                        output.collect(ikey,new FloatWritable(count/(sumcounts-count)));
                }
}
/**
 * Main method for running the Hadoop MapReduce job
 * @param args This main method takes no arguments
 * @throws Exception
  */
        public static void main(String[] args) throws Exception{
            int res = ToolRunner.run(new Configuration(), new MapRed(), args);
            System.exit(res);
        }
        public int run(String[] args) throws Exception{
                pass1();
                pass2();
                pass3();
                return 0;
        }
        public void pass1() throws Exception{
                // Job configuration
                JobConf conf= new JobConf(MapRed.class);
                conf.setJobName("Map-Reduce example");
                conf.setOutputKeyClass(Text.class);
                conf.setOutputValueClass(Text.class);
                conf.setMapperClass(XYIdentityMap.class);
                conf.setReducerClass(CountYReduce.class);
                conf.setInputFormat(TextInputFormat.class);
                conf.setOutputFormat(TextOutputFormat.class);
                // HDFS input and output directory
                conf.setInputPath(new Path("input"));
                conf.setOutputPath(new Path("intermediate0"));
                // Run map-reduce job
                JobClient.runJob(conf);
        }
        public void pass2() throws Exception{
                // Job configuration 2
                JobConf conf2= new JobConf(MapRed.class);
                conf2.setJobName("Map-Reduce example 2");
                conf2.setOutputKeyClass(Text.class);
                conf2.setOutputValueClass(CPair.class);
                conf2.setMapperClass(XYtoYXMap.class);
                conf2.setReducerClass(SplitReduce.class);
                conf2.setInputFormat(TextInputFormat.class);
                conf2.setOutputFormat(TextOutputFormat.class);
                // HDFS input and output directory
                conf2.setInputPath(new Path("intermediate0"));
                conf2.setOutputPath(new Path("intermediate1"));
                // Run map-reduce job
                JobClient.runJob(conf2);
        }
        public void pass3() throws Exception{
                JobConf conf3= new JobConf(MapRed.class);
                conf3.setJobName("Map-Reduce 3");
                conf3.setOutputKeyClass(Text.class);
                conf3.setOutputValueClass(IntWritable.class);
                conf3.setMapperClass(PairCollectMap.class);
                conf3.setReducerClass(NormalizationReduce.class);
                conf3.setInputPath(new Path("intermediate1"));
                conf3.setOutputPath(new Path("output"));
                JobClient.runJob(conf3);
        }
}
```

0.9.2 CPair.java

```
import java.io.*;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class CPair implements WritableComparable {
    private String x;
    private int count;
    // required methods
    public CPair(){
        x="";
        count=0;
    }
    public String toString(){
        return x + " " + count;
    }
    public void write(DataOutput out) throws IOException{
        out.writeChars(x + "\n");
        // "/n" needed here so that the readLine() calls in readFields will
        // read the correct input
        out.writeInt(count);
    }
    public void readFields(DataInput in) throws IOException{
        x = in.readLine();
        count = in.readInt();
    }
    public static CPair genc(StringTokenizer st){
        return new CPair(st.nextToken(), Integer.parseInt(st.nextToken()));
    }
    // real methods
    public CPair(String a, int b){
        x = a;
        count = b;
    }
    public int hashCode() {
        return x.hashCode()*127 + new Integer(count).hashCode();
    }
    public int compareTo(Object q){
        CPair p = (CPair)q;
        return getX().compareTo(p.getX());
    }
    public boolean equals(CPair p){
        return compareTo(p)==0;
    }
    public String getX(){
        return x;
    }
    public int getCount(){
        return count;
    }
}
```