Lab 7: MapReduce

Professor: Ronaldo Menezes

TA: Ivan Bogun

Department of Computer Science
Florida Institute of Technology

October 22, 2014

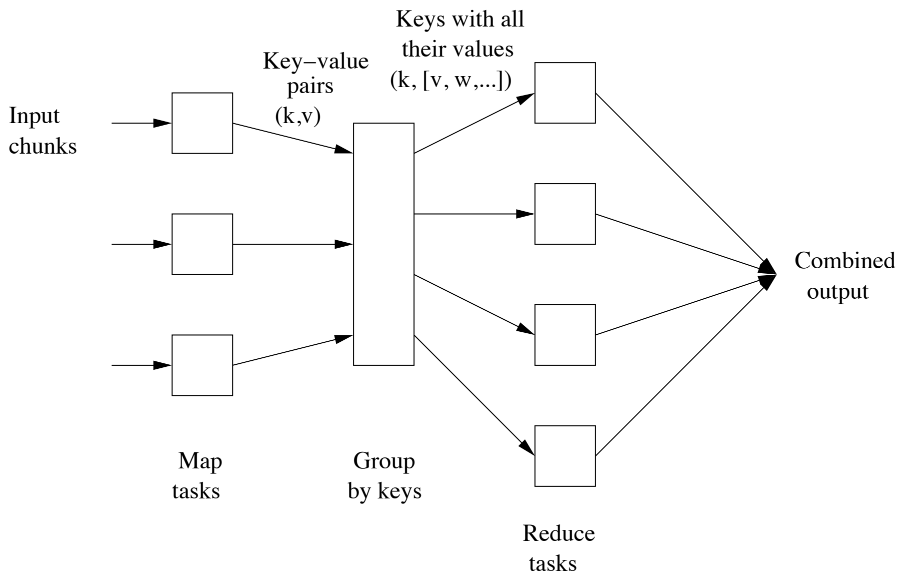

Figure 1: MapReduce diagram, as shown in [1]

1 General description

In this lab we will study hash tables with application to MapReduce. MapReduce is a programming model for processing large amounts of data via cluster, a collection of commodity computers. Contrary to parallel cluster computing doesn't require specialized hardware and scales to thousands or more computers. In this lab we will emulate clustering computing process using hash tables.

1.1 Overview

In order to process large amount of data it has to be split into chunks. Each chunk will be assigned to a specific computer to perform a *map* job. In this lab each chunk will be hashed to a specific computer using hashing function in order to uniformly distribute chunks (so that each map computer would do approximately same amount of computations). Note that in this lab you are not expected to implement map and reduce functions - they will be given. Once map jobs are finished we assign their result for computers to perform reduce jobs which finish computation. Fig. 1 shows the diagram. Example of *map* and *reduce* functions for word count can be found on wikipedia [1]

2 Implementation

Classes are expected to be implemented in the order they are presented.

[1]wiki on MapReduce

2.1 Generic Pair class

The following class *Pair.java* [2] demonstrates usage of generics in java. It will be used in the later stage, so nothing needs to be implemented here.

```java
// Pair.java

import java.util.Date;

public class Pair<Key, Value> {

    private final Key element0;
    private final Value element1;

    /**
     * Creates an instance of the pair given two objects
     * @param element0 first object in a pair
     * @param element1 second object in a pair
     */
    public Pair(Key element0, Value element1) {
        this.element0 = element0;
        this.element1 = element1;
    }

    public Key getFirst() {
        return element0;
    }

    public Value getSecond() {
        return element1;
    }

    public String toString(){
        String result="[";

        result+=element0.toString()+" - ";
        result+=element1.toString();

        result+="]";

        return result;
    }

    public static void main(String[] args) {
                Pair<String, Integer> pair1 = new Pair<String, Integer>("string",5);
                Pair<Date, Double> pair2 = new Pair<Date, Double>(new Date(),3.45);
                System.out.println(pair1);
                System.out.println(pair2);
        }

}
```

2.2 Generic LinkedList class

Implement generic *LinkedList.java* [3] class.

[2] Pair.java
[3] LinkedList.java

```java
import java.util.HashSet;
import java.util.Set;


public class LinkedList<Key, Value> {

   private int N;         // number of key-value pairs
   private Node first;    // the linked list of key-value pairs

   // a helper linked list data type
   private class Node {

      private Key key;
      private Value val;
      private Node next;     // pointer to the next Node

      public Node(Key key, Value val, Node next) {
         this.key = key;
         this.val = val;
         this.next = next;
      }
   }

   public void add(Key key, Value val) {
            // add new Key-Value pair to the linked list
   }

   // return number of key-value pairs
   public int size() {
      return N;
   }

   // is the symbol table empty?
   public boolean isEmpty() {
      return size() == 0;
   }

   public boolean contains(Key key) {
      // return true if the key is in the list, false otherwise
   }


   public Value get(Key key) {
             // return the value associated with the key, or null if the key is not present
   }

   public HashSet<Key> getKeys(){
      // return set of unique keys. Nothing to implement here
      Node firstNode=first;

      HashSet<Key> s = new HashSet<Key>();

      while(firstNode!=null){
            s.add(firstNode.key);
            firstNode=firstNode.next;
      }
      return s;
   }
```

```
    public void delete(Key key) {
        // remove key-value pair with given key
    }


    public String toString() {
              //return string representation of the list. Nothing to implement here.

        Node var = this.first;
        String result="";
        while (var != null) {
            result+=var.key.toString() + "\t \t " + var.val+"\n";
            var = var.next;
        }

        return result;
    }

    public static void main(String args[]) {
        LinkedList<String, Integer> list = new LinkedList<String, Integer>();
        list.add("asd", 5);
        System.out.println(list);
    }
}
```

2.3 Generic HashTable

Implement generic class *HashTable.java*[4] with collision resolution by chaining (using LinkedLists).

```
// HashTable.java
import java.util.HashSet;

public class HashTable<Key, Value> {

    private int N;                                          // number of elements in the
         hash table
    private int M;                                          // number of linked lists
    private LinkedList<Key, Value>[] lists; // Hash table with collision resolution by chaining

    public HashTable(int M) {
              // M is the number of linked lists the table should be initialized with
    }

    private int hash(Key key) {
        // multiplication method for hashing. Nothing to implement here.
        String strKey = key.toString();
        int intKey = 0;
        int strLength = strKey.length();

        final int RADIX=128; // See CLSR book for details

        for (int i =0; i <strLength ; i++) {
            intKey = (int)(Math.pow(RADIX, (strLength-1)-i)) * strKey.charAt(i) + intKey;
        }
        double A = (Math.sqrt(5) - 1) / 2;
```

[4]HashTable.java

```
        double res = intKey * A;
        res = res - Math.floor(res);
        int hashValue = (int) Math.floor(M * res);

        return hashValue;
    }

    public Value get(Key key) {
        // return Value, given key
    }

    public boolean contains(Key key){
        // true if key exists in the table, false otherwise
    }

    public void insert(Key key, Value val) {
        // insert Key-Value pair into hashtable
    }

    public void delete(Key key) {
        // delete key from the hashtable. Should only delete the first occurence of the key if
             there is more than one.
    }


    public HashSet<Key> getKeys(){
        // return HashSet of unique keys in the HashTable
    }

    public String toString() {
                // return string representation of the HashTable
    }
}
```

2.4 MapReduce for word count

MapReduce.java[5] interface with it's implementation for word count *WordCountMapReduce.java*[6] will be used later on. Nothing to implement here.

```
// MapReduce.java
import java.util.ArrayList;
public interface MapReduce<Key,Value> {

        public ArrayList<Pair<Key, Value>> map(ArrayList<Key> keys);
        public Pair<Key, Value> reduce(Pair<Key, ArrayList<Value>> list);
}
```

```
//WordCountMapReduce.java

import java.util.ArrayList;
import java.util.Iterator;

public class WordCountMapReduce implements MapReduce<String, Integer> {
```

[5]MapReduce.java
[6]WordCountMapReduce.java

```
        @Override
        public ArrayList<Pair<String, Integer>> map(ArrayList<String> keys) {
                // map function for word count. For every word in the list it will add the pair
                     <word,1> to the
                // result.

                ArrayList<Pair<String, Integer>> pairs = new ArrayList<Pair<String,
                     Integer>>(keys.size());

                for (Iterator<String> iterator = keys.iterator(); iterator.hasNext();) {
                        String value =iterator.next();
                        pairs.add(new Pair<String, Integer>(value, 1)); // <word,1>
                }
                return pairs;
        }

        @Override
        public Pair<String, Integer> reduce(Pair<String, ArrayList<Integer>> list) {
                // reduce function for word count. For every <word,list of integers> it will output
                     <word,sum(list of integers)>
                ArrayList<Integer> intList=list.getSecond();

                Integer sum=0;

                for (Iterator<Integer> iterator = intList.iterator(); iterator.hasNext();) {
                        sum +=iterator.next();
                }

                Pair<String, Integer> pair = new Pair<String, Integer>(list.getFirst(),sum);

                return pair;
        }

}
```

2.5 Combining all together

Use all of the previous classes to implement *WordCount.java*[7].

```
 import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;

public class WordCount {

        MapReduce<String, Integer> mapReduce;      // needed for map(), reduce() functions

        int numMappers;                                          // number of computers
             for map function
        int numReducers;                                         // number of computers
             for reduce function

        HashTable<String, Integer> mapHashTable;   // HashTable with <word,count>
        HashTable<Integer, Pair<String, ArrayList<Integer>>> reduceHashTable; // HashTable with
             <cpu,<word,list(word counts)>>

}
```

[7]WordCount.java

```java
public WordCount(int numMappers_, int numReducers_,
                MapReduce<String, Integer> mapReduce_) {
                        // constructor
}


public ArrayList<String> divide(String key) {
        // divide string into list of chunks( lines in this case ). Nothing to implement
             here.
        String[] tokens = key.split("\n");
        ArrayList<String> stringList = new ArrayList<String>(tokens.length);

        for (int i = 0; i < tokens.length; i++) {
                stringList.add(tokens[i]);
        }

        return stringList;
}

public HashTable<Integer,ArrayList<String>> assignMapJobs(String key) {

        // 1. divide key into chunks using divide() function
        // 2. assign each chunk to one of map computers enumerated from 0 to numMappers-1
        // starting from cpu=0;
        // for each line
        //              split it into words using
        //              String[] words = line.replaceAll("[^a-zA-Z ]",
             "").toLowerCase().split("\\s+");
        //              add <cpu,words> into resulting HashTable
        //
        //              increament cpu number using
        //              cpu=(cpu+1)% numMappers;


        return mapJobs;

}


public void processMapJobs(HashTable<Integer,ArrayList<String>> cpuVsWords){

        // process map jobs

        // for every unique cpu
        //
        //              Note: mapReduce.map() should be used here
        //
        //              for every word assigned to that cpu
        //                      add the pair <cpu,word> into mapHashTable
        //
}


public HashTable<Integer, Pair<String, ArrayList<Integer>>> assignReduceJobs(){

        // perform reduce step

        // get list of unique words in mapHashTable
        // starting from cpu=0;
        // for each word
        //              for as many times as word appears in the mapHashTable
```

```
            //                  add <word,list(ints)> into result
            //            add <cpu,pair(word,list(ints))> to the result
            //            cpu=(cpu+1)% numReducers;
            //
            // example:
            // assume mapHashTable has the following key-value pairs
            // <"cat",1>
            // <"cat",5>
            // <"cat",2>
            // then the following should be added: <cpu ID, pair("cat",list(1,5,2))>

      }

      public ArrayList<Pair<String, Integer>> processReduceJobs(
                  HashTable<Integer, Pair<String, ArrayList<Integer>>> reduceJobs){

            // process reduce jobs

            // for every unique cpu
            //
            //            Note: mapReduce.reduce() should be used here
            //
            //            for every <word,list(ints)>
            //                  add mapReduce.reduce(<word,list(ints)>) to the result
            //

}
```

3 Sample input-output

3.1 Input

Driver.java[8] with shows sample input.

```
// Driver.java
import java.util.ArrayList;

public class Driver {

      public static void main(String[] args) {

            WordCountMapReduce wordCountMapReduce = new WordCountMapReduce();

            int nMappers = 3;
            int nReducers = 2;
            WordCount wordCount = new WordCount(nMappers, nReducers,
                        wordCountMapReduce);

            String strToProcess = "one two three \n" + "three three one \n"
                        + "two three one \n" + "one three one \n";

            // get the table with <cpu #ID, words to process>. Should be of size nMappers.
            HashTable<Integer, ArrayList<String>> mapJobs = wordCount
                        .assignMapJobs(strToProcess);

            System.out.println(mapJobs);
```

[8]Driver.java

```
                // processJobs (populate wordCount.mapHashTable())
                wordCount.processMapJobs(mapJobs);

                // the table with <cpu #ID, <word,list(word counts)>>. Should be of size nReducers
                HashTable<Integer, Pair<String, ArrayList<Integer>>> reduceJobs = wordCount
                                .assignReduceJobs();

                System.out.println(reduceJobs);

                // list of <word,count>
                ArrayList<Pair<String, Integer>> counts = wordCount
                                .processReduceJobs(reduceJobs);
                System.out.println("Counts:");
                System.out.println(counts);
        }

}
```

3.2 Output

```
List #1
-----------------
1    [three, three, one]
-----------------
List #2
-----------------
0    [one, three, one]
0    [one, two, three]
-----------------
List #3
-----------------
2    [two, three, one]
-----------------

List #1
-----------------
1    [three - [1, 1, 1, 1, 1]]
1    [two - [1, 1]]
-----------------
List #2
-----------------
0    [one - [1, 1, 1, 1, 1]]
-----------------

Counts:
[[one - 5], [three - 5], [two - 2]]
```

4 Grade breakdown

basis	grade
Implementation	(60)
LinkedList()	15
HashTable()	15
WordCount()	30
Comments	(20)
General	10
Javadocs	10
Overall	(20)
Compiled	5
Style	5
Runtime	10
Total	100

References

[1] Rajaraman, Anand and Ullman, Jeffrey David, *Mining of massive datasets.* Cambridge University Press, 2011.