Aspect-Combining Functions for Modular MapReduce Solutions

Cristian Vidal Silva[1], Rodolfo Villarroel[2], José Rubio[3], Franklin Johnson[4], Érika Madariaga[5], Alberto Urzúa[6], Luis Carter[7], Camilo Campos-Valdés*[8], Xaviera A. López-Cortés[9]
[1]Ingeniería Civil Informática, Escuela de Ingeniería,
Universidad Viña del Mar, Viña del Mar, Chile
[2]Escuela de Ingeniería Informática, Facultad de Ing.,
Pontificia Universidad Católica de Valparaíso, Valparaíso, Chile
[3]Área Académica de Informática y, Telecomunicaciones,
Universidad Tecnológica de Chile, INACAP, Santiago, Chile
[4]Depto. Disciplinario de Computación e Informática,
Facultad de Ingeniería, Universidad de Playa Ancha, Valparaśo, Chile
[5]Ingeniería Informática, Facultad de Ingeniería,
Ciencia y Tecnología, Universidad Bernardo O'Higgins, Santiago, Chile
[6]Escuela de Kinesiología, Facultad de Salud,
Universidad Santo Tomás, Talca, Chile
[7]Ingeniería Civil Industrial, Facultad de Ingeniería,
Universidad Autónoma de Chile, Talca, Chile
[8]Programa Doctorado en Sistemas de Ingeniería, Facultad de Ingeniería,
Universidad de Talca, Curicó, Chile
[9]Depto. de Computación e Industrias, Facultad de Ingeniería,
Universidad Católica del Maule, Talca, Chile

***Abstract*—MapReduce represents a programming framework for modular Big Data computation that uses a function map to identify and target intermediate data in the mapping phase, and a function reduce to summarize the output of the map function and give a final result. Because inputs for the reduce function depend on the map function's output to decrease the communication traffic of the output of map functions to the input of reduce functions, MapReduce permits defining combining function for local aggregation in the mapping phase. MapReduce Hadoop solutions do not warrant the combining functioning application. Even though there exist proposals for warranting the combining function execution, they break the modular nature of MapReduce solutions. Because Aspect-Oriented Programming (AOP) is a programming paradigm that looks for the modular software production, this article proposes and apply Aspect-Combining function, an AOP combining function, to look for a modular MapReduce solution. The Aspect-Combining application results on MapReduce Hadoop experiments highlight computing performance and modularity improvements and a warranted execution of the combining function using an AOP framework like AspectJ as a mandatory requisite.**

***Keywords*—Combining; Hadoop; MapReduce; AOP; AspectJ; aspects**

I. Introduction

MapReduce represents a computation framework aiming to solve Big Data and Big Computation issues [1]–[4]. Hadoop is a MapReduce application tool [4], [5] with two main components, the Hadoop Distributed File System (HDFS) for an 'Infrastructural' point of view and MapReduce for the 'Programming' aspect. Hence, HDFS is a distributed and scalable file system designed for running on clusters of commodity hardware. HDFS follows the write-once, read-many approach to store huge files using streaming data access patterns to enable high throughput data access and simplifies data coherency issues [4], [5]. HDFS abstracts developers of distribution, coordination, synchronization, faults and failures, and supervision tasks details. Thus, developers must focus on two main computation functionalities: map and reduce.

Aspect-Oriented Programming (AOP) corresponds to a programming methodology for isolating crosscutting concerns functionalities and data to look for modular solutions [6]. Ideas of obliviousness and advisable classes appear in AOP. Wampler [7] indicates and demonstrates the AOP support and refinement of Object-Oriented Design (OOD) principle such as the Single Responsibility Principle (SRP) and Open-Closed Principle (OCP) mainly to remark the AOP practical benefits.

Even though MapReduce represents a framework to isolate a programmer of traditional faults and issues on traditional distributed programming approaches and frameworks, MapReduce demands to figure out solutions using their main two functions: map and reduce. Thus, these functions can include code out of their inner nature which are clear crosscutting concerns examples according to good modular programming and AOP principles [7].

Hadoop allows the definition of the combining function on the map output [5], [8], [9] to optimize the MapReduce framework functioning for local aggregation in the map phase, that is, a function to aggregate data in the map phase before sending them to the reduce phase. Even though the combiner function

is an optimization, Hadoop does not provide a guarantee of how many times it will call defined combining functions [8]. Thus, as a guarantee of combining execution, [8] proposed the use of the 'In-Mapper' Combining function, i.e., the combining function behavior directly inside the map function. Nonetheless, this solution does not respect object-oriented modularity principles such as the SRP [7], [10]. Looking for a modular application of the MapReduce programming framework, this article proposes and exemplifies the use of Aspect-Combining, an AOP application on MapReduce for the combining functions definition. Thus, the main contributions of this article are:

- Giving a review of performance and modularity issues of MapReduce combining solutions.
- Locating and justifying the presence of crosscutting-concerns in current optimal combining solutions.
- Defining and testing Aspect-Combining functions on classic case studies for getting more modular and usually more efficient results.
- Establishing the bases for future works about the symbiosis of Big Data and AOP solutions.

This article is organized as follows: Section II gives a description of the MapReduce framework and its main components. That section also explains the primary structure and principles of traditional AOP-AspectJ solutions. Section III reviews previous 'In-Mapper' Combining function and identifies crosscutting concerns issues to define Aspect-Combining functions. Section IV defines hypothesis and variables to measure in the experiments, and presents results of the use of Combining, 'In-Mapper' Combining, and Aspect-Combining proposal on a few application examples to highlight the main practical pros and cons of the Aspect-Combining function. Section V discusses validity of the established hypothesis. Section VI concludes and presents future research work.

II. MapReduce and AOP

A. MapReduce

MapReduce is a programming model proposed by Google [1]–[3] for distributed computation on massive amounts of data (Big Data), that is, MapReduce is an execution framework for large-scale data processing on clusters of commodity servers. MapReduce has already enjoyed widespread adoption by the use of Hadoop, a open-source implementation of MapReduce [5], [8].

MapReduce can refer to three concepts: 1) a programming model; 2) an execution framework to coordinates the execution of programs written in this programming style; 3) the implementation of 1) and 2), that is, MapReduce is the implementation of a programming model and its execution framework. Google is the proprietary of MapReduce implementation [1]–[3], and Hadoop is an open-source analogue substitute [5], [8].

Hadoop applies the Hadoop Distributed File System (HDFS), a highly fault-tolerant and distributed file system able to run on commodity hardware [4], [5], [8]. HDFS provides high throughput access to application data. HDFS is suitable for applications with large data sets such as the set of valid configurations in a Software Product Line (SPL) [9].

MapReduce basic idea is to partition a large problem into smaller sub-problems possibly independent able to run in parallel by different workers, that is, either by threads in a processor core, cores in a multi-core processor, multiple processors in a multi-processors machine, or many machines in a cluster [4], [8]. Fig. 1 shows the Hadoop functioning architecture. Hence, an iteration of a Hadoop solution (a.k.a job) normally executes in four steps: 1) Slicing to split the source data in multiples splices and deliver them to each map-worker or mapper. 2) Map to process the data (each mapper processes one or more chunks of data and sends the results to the shufflers). 3) Shuffle to organize the data. 4) Reduce to compact and write back results to the disk. Thus, intermediate results from each worker are then combined to yield the final output.

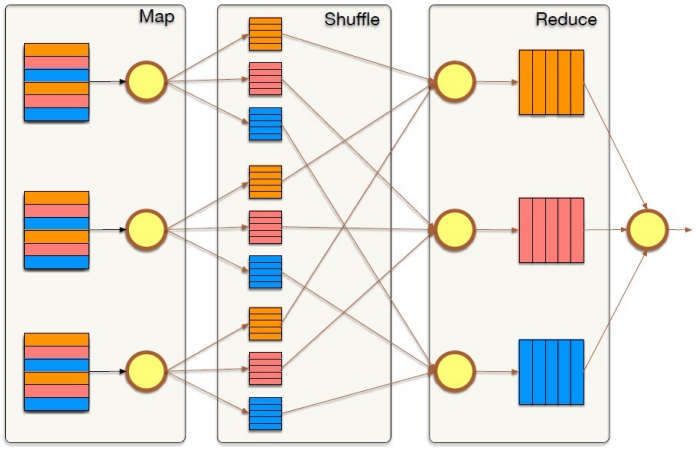

Fig. 1. MapReduce functioning architecture.

MapReduce allows for commutative and associative map functions to define combining function, that is, to decrease the amount of data shuffling between map-workers and reduce-workers [5], [8]. Combining functions work on the map functions output; hence, the output of combining functions represent the input of reduce functions. The MapReduce [1]–[3] execution framework coordinates functioning of map-workers and reduce-workers.

Hadoop solutions usually enable for the definition of a set of dependent jobs, i.e., the output of one job is used as an input for others and so on. Thus, a set of key-value records (K_{in}, V_{in}) is the input of a map function, and a $list(K_{inter}, V_{inter})$ corresponds to its output, that is, the input of a combining function if it were defined or input for the shuffling process. As was mentioned, the input for combining functions corresponds to the output of mappers, and the combining functions output will be the input for the shuffling process. Shuffling process orders and distributes data for reduce functions, that is, they get $(K_{inter}, list(V_{inter}))$ as input to produce an output (K_{out}, V_{out}) which can be the input of other map functions, and so on. Fig. 2 illustrates this described process.

B. Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [6] permits modularizing crosscutting concerns in base classes as aspects in Object-Oriented Programming (OOP). Aspects advise classes

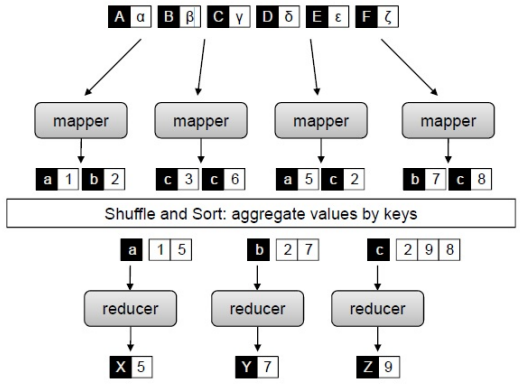

Fig. 2. A Simplified view of MapReduce.

statically in defined advisable modules and dynamically like events. AOP like AspectJ [6] defines oblivious advisable classes and modularizes crosscutting concerns as aspects, that is, orthogonal methods which are not part of the nature of advisable classes.

AOP well modularizes homogeneous crosscutting concerns as aspects [6], [7], [11]–[13]. However, aspects do not reflect the structure of refined features and the classes cohesion for the modularization of classes collaboration [14], [15]. Moreover, AOP languages like AspectJ [13], [16], [17] introduce implicit dependencies between aspects and advisable classes [18]–[21]. Hence, first, aspects do no respect the information hiding principle because oblivious classes can experience unexpected behavior and properties changes, and second, changes on the firm of advisable behavior can generate spurious and non-effective aspects. Thus, aspects need to know structure details about the advisable behavior and classes, a great issue for independent development.

Next, this article describes main AOP elements.

C. Join points and Pointcuts

A join point represents an event in the execution control flow of a program, that is, "a thing that happens" [13], [16]. Hence, in AOP [6], [11], a join point is a point of the program execution in which aspects advise advisable base modules. Examples of join points in AspectJ are method calls, method executions, object instantiations, constructor executions, field references and handler executions

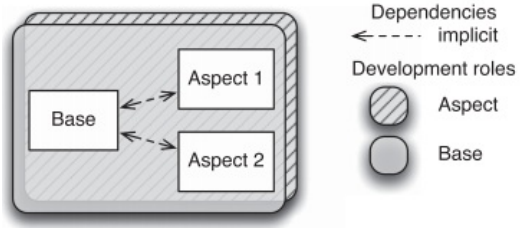

Fig. 3. AspectJ components and functioning.

```
public class HelloWorld {

      public static void main(String[] args){
        say("Hello");
        sayToPerson("Hello", "Cristian");
    }

   public static void say(String message) {
        System.out.println(message);
    }

    public static void sayToPerson(String message,
                                   String name) {
        System.out.println(name + ", " + message);
    }
}
public aspect BasicAspect {
  pointcut callSayMessage():
     call(public static void HelloWorld.say*(..));

  before() : callSayMessage() {
     System.out.println("Good day!");
  }

  after() : callSayMessage() {
     System.out.println("Thank you!");
  }
}
```

Fig. 4. AspectJ HelloWorld example.

According to [6], [7], a pointcut is a rule to pick out and define the join points occurrence and expose data from the execution context of those join points. Possible components of pointcut rules definition are call (method pattern), execution (method pattern), get (field pattern), set (field pattern), identifiers of time for advisable methods, objects associated to an advisable method, among others.

Just, for the pointcut definition in AOP like AspectJ languages of a method execution, two important times exist: when a methods is called (*call* time) and when a method is in execution (*execution* time). Furthermore, we can differentiate between *target* and *this* objects on the join point event, that is, the object whose method is in execution and the object that executes the method on the target object. Thus, *this* and *target* are the same object for pointcut rules of *execution* methods, and for *call* pointcut *this* is the object that order the *target* method execution.

D. Inter-type Declarations and Advices

In essence, inter-type declaration statically injects changes on fields, properties, and methods into existing advisable classes in AOP [6].

Advice defines crosscutting behavior regarding pointcut. Three type of advice in traditional AOP exist [6]: *before*, *after*, and *around* which determine how an advice runs at every picked out join point. These kinds of advice determine how the code injection works over the join points. Thus, in AOP like AspectJ languages there exist advice instances which run before their join points, run after their join points, and run in place of (or "around") their join points.

Fig. 3 [22] details the AspectJ components and functioning structure, that is, aspects advise oblivious base modules and they present implicit dependencies among them.

Fig. 4 [11] illustrates a basic AspectJ example, an advisable class *HelloWorld* and an aspect with two advice instances to inject behavior into the advisable class before and after calling a void method that starts with the word say in the class HelloWorld.

AspectJ mainly looks for of modular solutions and respecting modularity principles [16]–[18], [23]. This paper looks for getting modular MapReduce solutions by the use of AspectJ on Hadoop solutions.

III. Grouping Data Locally in MapReduce

The MapReduce computation in Hadoop does not require to put attention on embarrassingly-parallel issues such as synchronization and deadlock [7], [8]. Hadoop and MapReduce solutions possibly involve large data-intensive transferring from map-worker to reduce worker instances. Thus, since data transferring can be of a high cost; for the *local aggregation*, combining functions can considerably diminish the map output records with the same key in the map-workers.

```
public class WordCountMapper extends MapReduceBase
implements
Mapper<LongWritable, Text, Text, IntWritable>
{
  private final static IntWritable one =
          new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value,
    OutputCollector<Text, IntWritable> output,
    Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer tokenizer =
              new StringTokenizer(line);

      while (tokenizer.hasMoreTokens()){
         word.set(tokenizer.nextToken());
         output.collect(word, one);
      }
   }
}

public class WordCountReducer extends MapReduceBase
   implements
   Reducer<Text, IntWritable, Text, IntWritable>{

    public void reduce(Text key,
        Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output,
        Reporter reporter)
    throws IOException{
        int sum = 0;

        while (values.hasNext())
           sum += values.next().get();

      output.collect(key, new IntWritable(sum));
  }
}
```

Fig. 5. Traditional Hadoop WordCount example.

In practice, primary map and reduce functions in Hadoop [9], write intermediate results on local disk before sending them over the network. Those I/O processes possibly imply high computing and hardware costs depending on the network-latency and disk-space costs. Thus, using combining functions minimizes the amount of intermediate data transferring from map-workers to reduce-workers. That also allow decreasing the number and size of key-value pairs to shuffle from map-workers to reduce-workers for getting improvements on the MapReduce algorithmic efficiency. Just, combining functions are named "mini-reducers". In general, the use of combining functions seems adequate because map functions recognize intermediate-key and value pairs to send them for the shuffling and sorting process in traditional MapReduce solutions. The output of those processes corresponds to the input for reduce-worker instances.

```
private Text word = new Text();
HashMap<String, Integer> palabras =
        new HashMap<String, Integer>();

public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output,
        Reporter reporter) throws IOException{

  palabras.clear();

  String line = value.toString();
  StringTokenizer tokenizer = new StringTokenizer(line);

  while (tokenizer.hasMoreTokens()){
     String w = tokenizer.nextToken();

     if (palabras.containsKey(w))
         palabras.put(w, palabras.get(w )+ 1);

     else
         palabras.put(w, 1);
  }

  for (Entry<String, Integer> entry : palabras.entrySet()){
      String k = entry.getKey();
      Integer v = (Integer) entry.getValue();

      word.set(k);
      output.collect(word, new IntWritable(v));
  }
}
```

Fig. 6. 'In-Mapper' Combining Hadoop solution for the WordCount example.

A. Combining and 'In-Mapper' Combining

Even though map and reduce functions seem algorithmically simple to think and implement, combining function symbolize improvements performance for cases of high-traffic of data between map and reduce-workers. Combining functions act like the reduce functions [8] because they minimize the amount of intermediate data generated by each map-worker. For example, *WordCount* and *Average* represent two traditional solutions that support the use of a combining function, in the first case, functioning likes the reduce function. Nevertheless, combining functions execution are not always effective [5], [8]. Precisely, 'In-Mapper' Combining functions [8] solve those mentioned issues.

Fig. 5 presents a traditional Hadoop MapReduce solution for the *WordCount* example, and Fig. 6 shows an 'In-Mapper' Combining function to local aggregate data in the map phase and reduce the information traffic between map-worker and

```
public class Palabra {
    private String palabra;
    private Integer cuantas;

    public Palabra(String P){
        palabra = P;
        cuantas = 0;
    }

    public String getPalabra() {
        return palabra;
    }

    public void setPalabra(String palabra) {
        this.palabra = palabra;
    }

    public Integer getCuantas() {
        return cuantas;
    }

    public void incCuantas() {
        this.cuantas++;
    }
}
```

Fig. 7. Class Palabra for local aggregation in the Hadoop WordCount example.

reduce-worker. The input for that example corresponds to a set of words. Fig. 7 shows a new class *Palabra* for grouping values (local aggregation) in the *WordCount* example. The main function of the mapper functions in Fig. 5 and 6 look for identify words only, and to identify words and locally aggregate the already identified words count in the map function, respectively.

Fig. 8 shows the MapReduce solution for the *Average* example that looks for to obtain the average score of each student in a list of student and grade pairs. Fig. 9 shows an input example for the *Average* example. Fig. 10 illustrate the *GradeCount* class necessary for the local aggregation in the *Average* example.

Note that, for 'In-Mapper' Combining solution of the *WordCount* example, map function produces the same output as a traditional MapReduce solution, i.e., reduce function continues being the same. Nevertheless, as Lin and Dyer [8] illustrate, map and reduce functions of 'In-Mapper' Combining for the *Average* example do not produce and receive the same values such as those of the map and reduce functions in a traditional MapReduce solution of that example.

Even though the 'In-Mapper' Combining approach allows reducing information traffic from map-workers to reduce-workers [8], this approach implies to add more code and responsibilities on map functions. For example, 'In-Mapper' Combining of Fig. 6 includes a HashMap definition, and map function presents two actions in the loop, one to recognize each word and add them in the HashMap, and another one to update previous values of existing words; and map outputs these values after identifying all words and their occurrence number in the received input value. The number of sending and receiving messages of this solution would decrease if there were repeated words in the input. Nevertheless, map function grows in code and responsibilities, that is, the map function for 'In-Mapper' Combining approach is definitely lesser modular than its original version.

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class AverageMapper extends
  Mapper<LongWritable, Text, Text, IntWritable> {
  protected void map(LongWritable key, Text value,
    Context context) throws
    IOException, InterruptedException {

    String line = value.toString();
    StringTokenizer tokenizer = new
            StringTokenizer(line, ",");

    while(tokenizer.hasMoreElements()){
        String val = tokenizer.nextToken();

        String[] parts = val.split(" ");
        String part1 = parts[0];
        String part2 = parts[1];

        Text _id = new Text();
        _id.set(part1);
        Integer marks = new Integer(part2);

        context.write(_id, new IntWritable(marks));
    }
  }
}

import java.io.IOException;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class AverageReducer extends
Reducer<Text, IntWritable, Text, FloatWritable>{
    protected void reduce(Text key,
            Iterable<IntWritable> values,
            Context context)
    throws IOException, InterruptedException {
        Integer sum = 0;
        Integer cnt = 0;

        for (IntWritable value:values) {
            sum = sum + value.get();
            cnt = cnt + 1;
        }

        Float avg_m = (float) sum/cnt;

        context.write(key, new FloatWritable(avg_m));
    }
}
```

Fig. 8. Traditional Hadoop Average example.

```
Valery 6,Maria 4,
Paul 3,Paul 7,
Patrick 7,Fabrice 10,
Valery 10,Chris 0,
Fabrice 0,Chris 10,
Paul 10,Patrick 9,
Valery 2,Patrick 10,
Patrick 0,Sebas 9
```

Fig. 9. Input format for the Hadoop Average example.

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Writable;

public class GradeCount
            implements Writable{
    private IntWritable grade;
    private IntWritable count;

    public GradeCount() {
        this.grade = new IntWritable();
        this.count = new IntWritable();
    }

    //Custom Constructor
    public GradeCount(IntWritable grade,
                      IntWritable count) {
        this.grade = grade;
        this.count = count;
    }
   //Setter method to set the values
   //of a GradeCount object
   public void set(IntWritable grade,
                   IntWritable count){
      this.grade = grade;
      this.count = count;
   }

   //to get Count from GradeCount Record
   public IntWritable getCount()
   {
    return count;
   }

    @Override
    //overriding default readFields method.
    //It de-serializes the byte stream data
    public void readFields(DataInput in)
                    throws IOException {
      grade.readFields(in);
      count.readFields(in);
    }

    @Override
    //It serializes object data into
    //byte stream data
    public void write(DataOutput out)
                    throws IOException  {
     grade.write(out);
     count.write(out);
    }
 }
```

Fig. 10. Class GradeCount for local aggregation int the Hadoop Average example.

B. Aspect-Combining

Aspect-Combining represents a combining function as an AOP aspect on map function. In practice, such as Fig. 11 illustrate, AOP solutions would permit add behavior on MapReduce map methods just to isolate their functioning and nature. Thus, we propose Aspect-Combining. Aspect-Combining looks for the inclusion of structural and functioning elements of traditional 'In-Mapper' Combining functions. Hence, Aspect-Combining preserves the simplicity of the map function and guarantees the execution of the function combining. Furthermore, Aspect-Combining seems applicable by the use of any AOP approach over Hadoop. Next, this article describes a few AspectJ application examples.

Like for traditional combining function, the goal of Aspect-Combining is to locally aggregate data in map-worker instances to diminish the associated networking traffic in the map-workers for the shuffling process. Therefore, taking into account the components and functioning of the 'In-Mapper' Combining solutions such as those in Fig. 6; a class that contains the map function should also contain an attribute for local aggregation and methods for that process. Thus, in the *WordCount* example, it is necessary to know about each identified word and the number of previous occurrences of that word for updating its occurrences number. Hence, new attributes and methods for advisable classes are required by inter-type declaration in an AOP context. Likewise, in *Average* case, for each identified student, it would be necessary to sum their grades and also to count the number of their grades.

As Fig. 11 shows, three events exist for code injection in the advisable map method: before starting the execution of a map method to initialize attributes to group values, around the execution of a map method to group or create an identified element for local aggregation, and after the method map finishes for sending information to the next MapReduce step. Without considering the injection time for the occurrence of these events, pointcut rules are definable in AOP and AspectJ as well as the time for injecting the new behavior code that is analogue to the definition for AOP advices.

IV. Aspect-Combining Application and Results

A. Experiments

Table I shows the hypothesis and use of variables when conducting experimentation on classic Combining, In-Mapper Combining, and Aspect-Combining functions on the *WordCount* and *Average* examples.

For each experiment of Table I, the null hypothesis establishes that Aspect-Combining neither performs faster nor is more modular than classic Combining and 'In-Mapper' Combining solutions on the analyzed examples.

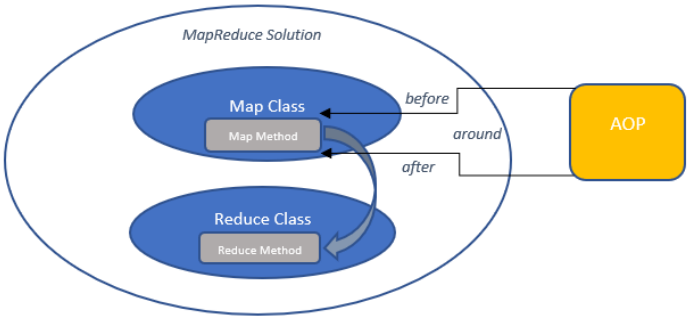

Fig. 11. Advisable map method of a MapReduce solution.

B. Results

In this section we discuss the results we obtained and how the null hypothesis has been rejected, thus accepting the alternative hypothesis.

We perform four experiments on the *WordCount* example and three experiments on the *Average* example to check the validity of the Aspect-Combining approach for modular MapReduce solutions in Hadoop.

Fig. 12 and 13 present the definition of pointcut instances for the Aspect-Combining of the *WordCount* and *Average* examples, in this case, 3 point cuts for each case: one to start collecting data, one for the collect and write methods call inside the advised map methods, and one for the end of the map method execution.

Fig. 14 and 15 show Aspect-Combining inter-type declaration for the WordCount Average examples to add and manipulate the required object collection, ArrayList of class Palabra and HashMap of class GradeCount instances, respectively.

Finally, for Aspect-Combining in the *WordCount* and *Average* examples, Fig. 16 and 17 present advice instances for before the method map execution to initialize the attribute for local aggregation, around the grouping values process, and after the execution of map method to effectively send the locally grouped values to the next MapReduce step.

As a practical functioning and results evaluation, Tables II and III present traditional In-Mapper and Aspect-Combiner results for the *WordCount* and *Average* examples to appreciate and compare them. We run practical experiments in a single Lenovo ThinkPad Edge E530 laptop of 2.50 GHz, 16GB of RAM and a Core i3 processor. For the *WordCount* examples, as input files, Words is a text file of 168 bytes, and ebook is a file of 1.6 MB; whereas for the *Average* examples, input files were generated taking in account ten students, and grades from 0 to 100.

Although these experiments did not run in a cluster of computing machines, and knowing the main practical improvement of 'In-Mapper' and Aspect-Combining is a reduction of traffic between map-workers and reduce-workers; surprisingly, Aspect-Combining permits obtaining better modularity and better performance for big-input examples. Hence, only for a single and small file, the traditional *WordCount* solution without combining approaches obtains the best time. In the *WordCount* example, for two files, 'In-Mapper' *WordCount* solution is the best and Aspect-Combining the 2nd one. For all other cases, Aspect-Combining presents the best performance. Thus, in addition to the best modularity, Aspect-Combining permits getting efficient computation results in one machine execution. This situation would be the same cluster environments.

V. Discussion

The null hypothesis establishes that Aspect-Combining does not improve the modularity for the presence of crosscutting-concerns issues and the execution-time compared to the Combining and 'In-Mapper' Combining solutions for testing on the *WordCount* and *Average* examples. To refute that hypothesis, we review the modular code of Aspect-Combining solutions in which the map function has only one responsibility, and we analyze the execution-time for experiments described in Table II and Table III, both tables for random files of different sizes. For the appreciated results, we accepted the alternative hypothesis that the Aspect-Combining

```
pointcut init(WordCountMapper mapper):
    execution(* map(..)) && target(mapper);

pointcut send(WordCountMapper mapper, Text word,
    IntWritable val):
    call(* collect(..)) && args(word, val) &&
    this(mapper)
    && !within(AspectMapper);

pointcut end(WordCountMapper mapper, LongWritable key,
    Text value, OutputCollector<Text, IntWritable> output,
    Reporter reporter): execution(* map(..)) &&
    args(key, value, output, reporter) &&
    target(mapper);
```

Fig. 12. Pointcut definition for Aspect-Combining in the WordCount example.

```
aspect AspectMapper {
    pointcut init(AverageMapper mapper):
        execution(* map(..)) && target(mapper);

    pointcut send(AverageMapper mapper, Text st,
        GradeCount gc):
        call(* write(..)) && args(st, gc) &&
        this(mapper)
        && !within(AspectMapper);

    pointcut end(AverageMapper mapper, LongWritable key,
        Text value, Context context): execution(* map(..)) &&
        args(key, value, context) &&
        target(mapper);
```

Fig. 13. Pointcut definition for Aspect-Combining in the Average example.

outperforms the traditional approach of Combining and 'In-Mapper' Combining solutions on the *WordCount* and *Average* case-studies.

The SRP establishes that each module or class should have one and only one purpose and reason to change since if a class has more than one responsibility, then the responsibilities become coupled [10]. According to [7], "The SRP is the OOD solution to the classic 'separation-of concerns' problem". Thus, Aspect-Combining permits simple map functions and efficient MapReduce solutions, even though, for the weaving process of AOP, the code of 'In-Mapper' Combining and the final one of Aspect-Combining should be equivalent.

```
private ArrayList<Palabra> WordCountMapper.palabras;

public void WordCountMapper.initPalabras(){
    palabras = new ArrayList<Palabra>();
}

public void WordCountMapper.incPalabra(String palabra){
    if (palabras.contains(palabra)){
        Integer index = palabras.indexOf(palabra);
        Palabra p = palabras.get(index);
        p.incCuantas();
        palabras.set(index, p);
    }
    else
        palabras.add(new Palabra(palabra));
}

public ArrayList<Palabra> WordCountMapper.getPalabras(){
    return palabras;
}
```

Fig. 14. Inter-type declaration for Aspect-Combining in the WordCount example.

TABLE I. HYPOTHESES AND DESIGN OF EXPERIMENTS FOR WORDCOUNT AND AVERAGE MAPREDUCE EXAMPLES

Hypotheses of Experiments 1 and 2	
Null Hypothesis (H_0)	Aspect-Combining solutions neither are faster nor more modular than Combining and 'In-Mapper' Combining for the *WordCount* case-study.
Alt. Hypothesis (H_1)	Exist cases in which Aspect-Combining solutions performs faster than Combining and In-Mapper Combining and does not present crosscutting concerns for the *WordCount* case-study.
files used as input	Randomly generated files of words of name and grade. Size of files are from 10KB to 1MB.
Blocking variables	In each experiment, we generated a set of files in increasing size.
Hypotheses of Experiment 2	
Null Hypothesis (H_0)	Aspect-Combining solutions neither are faster nor more modular than Combining and 'In-Mapper' Combining for the *Average* case-study.
Working Hypothesis (H_1)	Exist cases in which Aspect-Combining solutions perform faster than Combining and 'In-Mapper' Combining, and Aspect-Combining solutions do not present crosscutting concerns for the *Average* case-study.
Files used as input	Randomly generated files of pairs of name and grade. Size of files are from 10KB to 1MB.
Blocking variables	In each experiment, we generated a set of files in increasing size.
Constants	
Hadoop 2.4.1 in Ubuntu Linux 14.02	*WordCount and Average solutions implemented in 2016 and 2017, respectively*

TABLE II. WORDCOUNT SOLUTIONS - PRACTICAL EVALUATION

Input	Traditional WordCount	In-Mapper WordCount	Aspect-Combiner Word-Count
Words file (168B)	**2768867581 ns**	2797626514 ns	2830724641 ns
Words file (168B)+ ebook (1.6MB)	6474883019 ns	**4750820118 ns**	5675306443 ns
Words file (168B) + 30 ebook copies (48MB)	36835481761 ns	36011355288 ns	**33913147695 ns**
Words file (168B) + 50 ebook copies (80MB)	52135395542 ns	58132499534 ns	**51053756385 ns**

TABLE III. AVERAGE SOLUTIONS - PRACTICAL EVALUATION

Input	Traditional Average	In-Mapper Average	Aspect-Combiner Average
100 files (48.1KB)	12208547623 ns	**11833383768 ns**	13055207744 ns
200 files (192.6MB)	139049238451 ns	**61734379340 ns**	68603119714 ns
400 files (384.1MB)	257664874145 ns	134202559848 ns	**129013569364 ns**

```
private Map<String, GradeCount> AverageMapper.scores;

public void AverageMapper.initScores(){
    scores  = new HashMap<String, GradeCount>();
}

public void AverageMapper.addScore(String st, GradeCount gc){
    if (scores.containsKey(st)) {
        GradeCount ss = (GradeCount) scores.get(st);

        Integer sum = Integer.parseInt(ss.getGrade().toString()) +
                      Integer.parseInt(gc.getGrade().toString());
        Integer count  = Integer.parseInt(ss.getCount().toString()) +

        ss.set(new IntWritable(sum), new IntWritable(count));

    } else {
        scores.put(st, gc);
    }
}

public Map<String, GradeCount> AverageMapper.getScores(){
    return scores;
}
```

Fig. 15. Inter-type declaration for Aspect-Combining in the Average example.

```
before(WordCountMapper mapper): init(mapper){
      mapper.initPalabras();
}


void around(WordCountMapper mapper, Text word,
     IntWritable val): send(mapper, word, val){

    mapper.incPalabra(word.toString());
}

after(WordCountMapper mapper, LongWritable key, Text value,
    OutputCollector<Text, IntWritable> output,
    Reporter reporter) throws IOException:
    end(mapper, key, value, output, reporter){

    ArrayList<Palabra> palabras = mapper.getPalabras();

    for(Palabra p: palabras){
         Text w =  new Text();
         IntWritable c = new IntWritable();

         w.set(p.getPalabra());
         c.set(p.getCuantas());

         output.collect(w, c);
    }
}
```

Fig. 16. Advices for Aspect-Combiner in the WordCount example.

VI. CONCLUSIONS

In this section, we present the lessons we learned while developing the Aspect-Combining solutions:

- Aspect-Combining presents a practical symbiosis between MapReduce and AOP. In particular, this article presented a Hadoop and AspectJ for the implementation of Aspect-Combining.

```
before(AverageMapper mapper): init(mapper){
      mapper.initScores();
}
void around(AverageMapper mapper, Text st,
     GradeCount val): send(mapper, st, val){

    mapper.addScore(st.toString(), val);
}
after(AverageMapper mapper, LongWritable key,
      Text value, Context context)
      throws IOException, InterruptedException:
end(mapper, key, value, context){
  Map<String, GradeCount> sts = mapper.getScores();
  Iterator<Map.Entry<String, GradeCount>> itr1 =
                          sts.entrySet().iterator();
  while (itr1.hasNext()) {
      Entry<String, GradeCount> entry1 =
                                itr1.next();
      String s_id_1 = entry1.getKey();
      Text t = new Text(); t.set(s_id_1);
      GradeCount ss = entry1.getValue();
      context.write(t, ss);
  }
}
```

Fig. 17. Advices for Aspect-Combiner in the Average example.

- Thinking on the primary functions of MapReduce along with their focus, original combining functions are usually adequate to preserve the map function nature and simplicity. Nonetheless, this article pointed out its non-effectiveness and cost. Therefore, 'In-Mapper Combining seems more practical, but they do not respect modularity principles. Hence, this article presented and practically proved the benefits of Aspect-Combining for modular MapReduce solutions and, for big data-input, possible more efficient results than Combining and 'In-Mapper' Combining Hadoop solution.
- Although a class for map-worker permit the producion of modular solutions, a programmer is in charge of putting attention on Initializer, Map, and Close methods, that is, setup(..), map(..), and cleanup(..) methods in Hadoop which does not permit an independent development. Thus, Aspect-Combining approach separates these functions as advice instances, and the map-worker focuses only on an oblivious map(..) method of before(..), around(..) and after(..) advice instances which operate similar to Initialize, Map, and Close methods of Fig. 18 [8].

```
class Mapper
    method Initialize
        H ← new AssociativeArray
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
    method Close
        for all term t ∈ H do
            Emit(term t, count H{t})
```

Fig. 18. A modular structure of Mapper class in MapReduce Solutions.

As future work, this research group plans to review more about AOP on MapReduce applications to figure out the applicability of other AOP practical approaches such as JPI [18], [20], [22], [24] and Ptolemy [17] on Hadoop [5], [8] and Giraph approaches [25], and compare their effectiveness and practical performance. Giraph also permits defining combining functions without a guarantee for their execution [25], and Aspect-Combining seems adequate to guarantee their execution.

ACKNOWLEDGMENT

This work was partially supported by CONICYT-CCV/Doctorado Nacional/2018-21181055.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[2] ——, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[3] ——, "Mapreduce: A flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1629175.1629198

[4] D. Miner and A. Shook, *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*, 1st ed. O'Reilly Media, Inc., 2012.

[5] T. White, *Hadoop: The Definitive Guide*, 4th ed. O'Reilly Media, Inc., 2015.

[6] G. Kiczales, "Aspect-oriented Programming," *ACM Comput. Surv.*, vol. 28, no. 4es, dec 1996. [Online]. Available: http://doi.acm.org/10.1145/242224.242420

[7] D. Wampler, "Aspect-oriented design principles: Lessons from object-oriented design," *Proceedings of the Sixth International Conference on Aspect-Oriented Software Development*, vol. AOSD'07, pp. 615–636, 2007.

[8] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010.

[9] J. A. Galindo, M. Acher, J. M. Tirado, C. Vidal, B. Baudry, and D. Benavides, "Exploiting the enumeration of all feature model configurations: A new perspective with distributed computing," in *Proceedings of the 20th International Systems and Software Product Line Conference*, ser. SPLC '16. New York, NY, USA: ACM, 2016, pp. 74–78. [Online]. Available: http://doi.acm.org/10.1145/2934466.2934478

[10] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[11] R. Laddad, *AspectJ in Action: Enterprise AOP with Spring Applications*, 2nd ed. Greenwich, CT, USA: Manning Publications Co., 2009.

[12] R. Miles, *AspectJ Cookbook*. O'Reilly Media, Inc., 2004.

[13] J. D. Gradecki and N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[14] S. Apel, D. Batory, C. Kstner, and G. Saake, *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.

[15] S. Apel, D. Batory, and M. Rosenmüller, "On the Structure of Cross-cutting Concerns: Using Aspects or Collaborations?" Oct. 2006.

[16] G. Kiczales and M. Mezini, "Aspect-oriented programming and modular reasoning," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 49–58. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062482

[17] H. Rajan, G. T. Leavens, R. Dyer, and M. Bagherzadeh, "Modularizing crosscutting concerns with ptolemy," in *Proceedings of the Tenth International Conference on Aspect-oriented Software Development Companion*, ser. AOSD '11. New York, NY, USA: ACM, 2011, pp. 61–62. [Online]. Available: http://doi.acm.org/10.1145/1960314.1960332

[18] E. Bodden, E. Tanter, and M. Inostroza, "Join Point Interfaces for Safe and Flexible Decoupling of Aspects," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 7:1–7:41, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2559933

[19] C. Vidal Silva, R. Villarroel, R. Schmal, R. Saens, C. Del Rio, and T. Tigero, "Aspect-Oriented Formal Modeling: (AspectZ + Object-Z) = OOAspectZ," *COMPUTING AND INFORMATICS*, vol. 34, no. 5/15, 2015.

[20] C. Vidal Silva, R. Villarroel, L. López, M. Bustamante, R. Schmal, and V. Rea, "JPI UML Software Modeling: Aspect-Oriented Modeling for Modular Software," *International Journal of Advanced Computer Science and Application IJACSA*, vol. 6, no. 12, 2015.

[21] E. Bodden, "Closure Joinpoints: Block Joinpoints Without Surprises," ser. AOSD '11. New York, NY, USA: ACM, 2011, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/1960275.1960291

[22] C. Vidal Silva, R. Saens, C. Del Rio, and R. Villarroel, "Aspect-Oriented Modeling: Applying Aspect-Oriented UML Use Cases and Extending Aspect-Z," *COMPUTING AND INFORMATICS*, vol. 32, no. 3, 2013.

[23] ——, "OOAspectZ and aspect-oriented UML class diagrams for Aspect-oriented software modelling (AOSM)," *INGENIERIA E INVESTIGACION*, vol. 33, no. 3, 2013.

[24] C. Vidal Silva, R. Villarroel, and C. Pereira, "JPIAspectZ: A formal specification language for Aspect-Oriented JPI applications," *Proceedings of XXXIII International Conference of the Chilean Computer Science Society*, 2014.

[25] R. Shaposhnik, C. Martella, and D. Logothetis, *Practical Graph Analytics with Apache Giraph*, 1st ed. Berkely, CA, USA: Apress, 2015.