This paper is a post-print paper accepted in “International Conference on Future Internet of Things and Cloud (FiCloud), 2014“

The final version of this paper is available through IEEE Xplore in the next link: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6984223

Moran, J.; De La Riva, C.; Tuya, J., "MRTree: Functional Testing Based on MapReduce's Execution Behaviour," in Future Internet of Things and Cloud (FiCloud), 2014 International Conference on , vol., no., pp.379-384, 27-29 Aug. 2014

doi: 10.1109/FiCloud.2014.67

IEEE copyright notice. © 2014 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

MRTree: Functional Testing based on MapReduce´s execution behaviour

Jesús Morán, Claudio de la Riva, Javier Tuya
Department of Computing
University of Oviedo
Gijón, Spain
moranjesus@lsi.uniovi.es, claudio@uniovi.es, tuya@uniovi.es

***Abstract*—MapReduce is a paradigm that allows parallel processing of large amounts of data. MapReduce programs combined with their underlying run-time framework have distinctive features that are prone to include unexpected behaviors not present in other types of programs. This paper describes an approach to functional testing of MapReduce programs based on a hierarchical classification of a number of potential faults that may occur in MapReduce programs over Hadoop. This classification, called MRTree, is then used to derive test cases able to detect the faults represented in MRTree and illustrated with some examples.**

Keywords— Testing, MapReduce, Big Data, Hadoop

I. INTRODUCTION

The MapReduce programming paradigm [1] describes a parallel programming model based on "divide and conquer". One of its scopes is Big Data, where massive amounts of structured and unstructured information are handled.

A MapReduce program is usually implemented on a framework that deals with the management of parallelization and network fault management, among others tasks. These frameworks have a particular way of executing programs: for example, they split the program input into several parts, sending them to different computers, schedule which computers performed the first analysis, etc. Due to the new paradigm, the constraints imposed by the framework and the possible failures that can occur in any program, we have new specific potential failures when combining MapReduce with the framework.

The main contributions of this paper are (1) the identification a number of specific faults that may be introduced into a MapReduce program, and (2) the approach for functional testing of these programs in order to reveal the failures produced by these faults. This is illustrated using two different examples.

The principles of the MapReduce paradigm and the execution flow that the program follows are summarized in Section II. Subsequently, Section III contains an overview of issues that influence the MapReduce level and a state of the art. Section IV identifies functional faults that may occur in a MapReduce program. For some of the faults, in Section V, the tests are derived by introducing a program and test criteria.

II. PRINCIPLES OF MAPREDUCE

The MapReduce programs start with a problem which is divided into several parallel sub-problems to solve. The programming model consists mainly of a *Map* function and a *Reduce* function whose inputs and outputs are pairs <key, value>. The keys and values can be composed of several elements, which in this article we will be called attributes

The *Map* function emits zero or more pairs <key, value>, where each unique key represents a sub-problem to be solved, and the value represents the information needed to solve it. The *Reduce* function receives pairs <key, list[values]> obtained from processing different *Map* functions, where key represents the sub-problem identifier and the list represents the information needed to solve it.

First, several *Map* functions are executed in parallel, each of which analyzes a different part of the program input. Then several *Reduce* functions are executed, also in parallel. The tasks of management of network problems, efficient use of resources and others, are performed by a framework, usually Hadoop [2].

In addition to the *Map* and *Reduce* functions, there may exist other functions, such as the *Combine* function, whose goal is to decrease the amount of information that the *Reduce* function receives. The *Combine* function is located between the *Map* output and the *Reduce* input. This function receives pairs of <key, list[subset values]> as input, i.e., a key that identifies a sub-problem and some information which that sub-problem needs. The *Combine* function pre-processes information and emits <key, value> pairs that the *Reduce* function uses to continue the execution cycle. Hadoop does not guarantee whether *Combine* will be executed, and if so, it is not known with certainty how many pairs <key, value> it receives as input [3].

For example, a program that obtains the maximum temperature based on historical information from weather stations, may implement the three functions described above:

- *Map* function: Receives <year, temperature> pairs, performs validations and emits them as pairs <year, temperature>.
- *Combine* and *Reduce* functions: Receive <year, list[temperatures]> pairs, obtain the maximum

temperature and emit pairs <year, maximum temperature>.

An example of program execution in Hadoop is shown in Fig. 1 for year 1990 with 1º, 3º and 10º; 1991 with 15º, and 1992 with 1º. The execution is performed on three computers, two of them initially run the *Map-Combine* functions in parallel. The total key space is divided into several disjoint sets, which will be assigned to computers that will execute the *Reduce* functions. Each pair <key, value> emitted by the *Map-Combine* functions is sent to the computer to which that key is assigned. Among others, the computer 3 has assigned keys 1990 and 1991, whereas the Computer 2 has assigned the key 1992. Finally, it runs each of the *Reduce* functions and the program result is obtained.

Other functions that interact with the program may exist (Partitioner, Sort, etc.). In addition, a Hadoop program may be composed of a concatenation of multiple *Map-Reduce-Combine* functions. This paper consider programs that include *Map*, *Reduce* and *Combine* without concatenations.

III. MAPREDUCE TESTING

Big Data programs testing is classified by Gudipati et al. [4] into "Validation of Pre-Hadoop processing", "Validation of Map Reduce Process", "Validation of Data Extract, and Load into EDW (Enterprise Data Warehouse)", "Validation of Reports", "Performance Testing", "Failover tests" and "Big Data Test infrastructure design". Some of these types of testing have been addressed previously in the literature and covered by some tools. For example, unit testing can be performed with MRUnit library [5] by setting the input and output that should take in MapReduce functions.

The performance of a Hadoop program depends in part on the configuration of both the program and the cluster [6]. To test the performance of Hadoop, cluster benchmarks such as MRBench [7] HiBench [8] or GridMix [9] may be used or also monitoring tools to detect bottlenecks, slow computers, etc. Testing of the environment setup is studied in Bigtop [10], which is a project to test interoperability with other Hadoop projects.

In the "Validation of Map Reduce Process" category, Dorre et al. [11] perform static checking of the data types of the pairs <key, value> sent and received by the MapReduce functions. Other approaches are based in symbolic execution. This requires a framework that must be adapted to analyze different

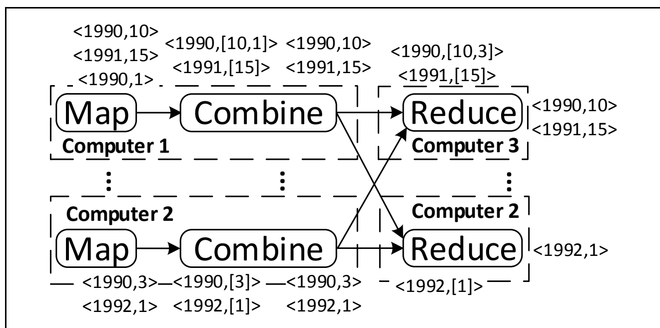

Fig. 1. Example of running MapReduce

code paths and some program requirements established by the tester [12]. The approach of this paper is similar, but it addresses the tests from the standpoint of identification of potential functional faults, in order to derive repeatable tests that may be designed at an early stage, before program implementation.

IV. MRTREE: A CLASSIFICATION OF FAULTS IN MAPREDUCE PROGRAMS

A potential failure is a possible discrepancy between the program's desired output and the output produced. The fault (defect) that causes the failure may be detected or masked if the program has different behavior depending on run-time conditions. In order to identify these potential failures, we elaborate a hierarchical classification called MRTree (MapReduce Tree).

MRTree represents a fault classification that allows us to identify specific functional failures of programs developed using the MapReduce paradigm and the underlying Hadoop framework. The goal is to derive tests to detect each potential fault represented by MRTree, as we will see in Section V.

MRTree is represented as a tree whose nodes represent potential faults that are hierarchically organized. When a parent node has several child nodes, it means that the failures of the child nodes can occur simultaneously, unless it is represented by an XOR connector. The leaf nodes, henceforth Fault-Nodes, have an identifying name, and a label (in the edge) with information about the function or functions associated with the fault. Characters M, C and R are used for the *Map*, *Combine* and *Reduce* functions, respectively. Thus labels add a third dimension to the tree, as each Fault-Node can be applicable for one or more of these functions. Fig. 2 depicts the MRTree that will be described in this paper. Next, we describe the Fault-Nodes of MRTree, organized by the top-level nodes of the tree.

A. Specific MapReduce problems

MapReduce's specific failures are those that can occur due to the distinctive features of execution in Hadoop. The faults may be caused by a wrong program design and/or wrong implementation. Fault-Nodes in this category are described below. In section V we illustrate how the tests are derived for each of the Fault-Nodes.

URIO (*Unexpected Results Input Order*): In the production environment it is difficult to guarantee the execution order of each function *Map* (inputs are executed in parallel, some of these have to be reprocessed because computers can fail, etc.). The *Combine* or *Reduce* functions may produce unwanted results if the input values are in an unexpected order. On other hand, the development environment can consist of a single computer that should not be fail during the test and does not always execute inputs in parallel. This implies that in development it is more likely to produce repeatable outputs than production, which may affect the program's result.

For example, the *Reduce* function has a fault if it assumes that input values will be ordered increasingly, but this order is not preserved by the previous functions.

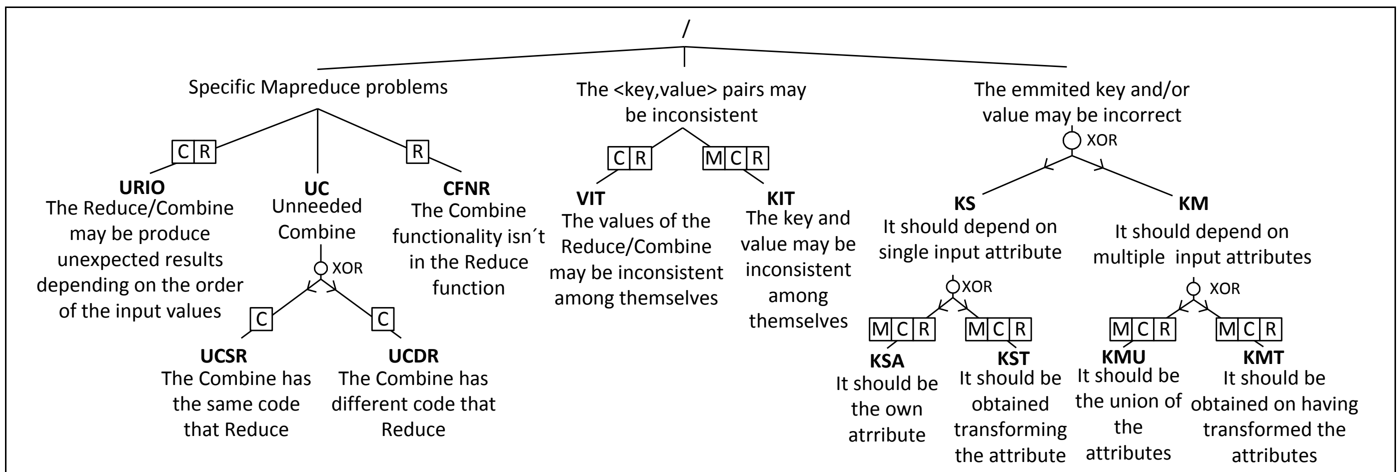

Fig. 2. MRTree, a classification of faults in MapReduce

UC (*Unneeded Combine*)**:** The *Combine* function takes a pre-processed subset of the <key, value> pairs emitted by the *Map* function. Whether *Combine* is executed or not, depends on many factors: size of program entry, time to process inputs, time to create <key, value> pairs in the *Map* function, etc. The *Combine* function is sometimes executed with one subset of information and at other times with a different one based on: the size of a buffer that stores the pairs <key, value> emitted by the *Map* function, the buffer threshold from which these pairs <key, value> are extracted to continue the cycle of execution, time spent emptying the buffer, etc. The different inputs of the *Combine* function are a potential cause for failures.

Not all programs support the *Combine* function, but when the *Reduce* function is commutative and associative [1] the program allows a *Combine* function with the same implementation as the *Reduce* function, and vice versa. Under other conditions the program can accept a *Combine* function having different implementation than the *Reduce* function, which allows us to identify two different situations: **UCSR** (*Unneeded Combine Same Reduce*) when the Combine function has the same implementation as the *Reduce* function, and **UCDR** (*Unneeded Combine Different Reduce*) when they have different implementations.

CFNR (*Combine Functionality Not in Reduce*): The *Reduce* function must be designed under the assumption that the information that it receives may or may not have been pre-processed by the *Combine* function If a <key, value> pair is not processed by the *Combine* function, *Reduce* must properly process that input. In other words, the pre-processing that the *Combine* function performs must also be done by the Reduce function. Although the *Combine* function may perform correctly its pre-processing, this functionality may not be properly implemented by the *Reduce* function, thus leading to potential failures.

B. The <key,value> pairs may be inconsistent

During MapReduce execution, unexpected situations may generate inconsistent <key, value> pairs. Inconsistency refers to when each key and each value taken in isolation is computed correctly on any *Map*, *Combine* or *Reduce* functions, but in the program as a whole their values are not compatible. We consider two types of inconsistencies. Later, Section V describes how to derive tests for each Fault-Node.

VIT (*Values Inconsistent Themselves*): It happens when a *Combine* or *Reduce* function receives more than one inconsistent value for the same key, although independent pairs <key, value> emitted by *Map* function are correct. For example, if the input of *Reduce* function is a pair <year, list[temperature information]> such that <1900, ["minimum 20", "maximum: 10"]>, there is an inconsistency because the maximum temperature cannot be lower than the minimum. The *Map* functions could have successfully emitted pairs <1900," minimum: 20"> and <1900, "maximum: 10">, but in the *Reduce* function the pairs with key 1900 are inconsistent.

KIT (*Key Value Inconsistent Themselves*): It happens in *Map*, *Combine* or *Reduce* functions when a key and its value are correctly computed, but the whole <key, value> is incoherent. For example, if the *Map* function reads from multiple databases, and emits pairs <id_father, id_son> such as <123, 123> there is an inconsistency. The *Map* function could have obtained properly from databases that the key is 123 and the value is 123, but really the pair <id_father, id_son> is incoherent.

C. The emmited key and/or value may be incorrect

This fault happens when a *Map*, *Combine* or *Reduce* function must emit a <key, value> pair, and it emits a different and incorrect pair due to faulty implementation. Because these faults are similar to those occurring in other paradigms they are briefly described and illustrated by a simple example.

KSA (*Key-value Single Attribute*): A key and/or value that should have the content of one input attribute is incorrectly emitted. Consider a program that obtains the maximum temperature by year. The *Map* function receives pairs <year, temperature> and should emit them to the *Reduce* function to compute the maximum temperature. If *Map* function does not emit negative temperatures, it has a fault.

KST (*Key-value Single Transforming*): A key and/or value that should be obtained after processing a single input attribute is incorrectly emitted. Consider a program that receives as input a year and the temperature in degrees Celsius and

calculates the average temperature in degrees Fahrenheit. The *Map* function receives pairs <year, temperature in Centigrade degrees> and emits pairs <year, temperature in Fahrenheit degrees>, then the *Reduce* function computes the average. In a *Map* function a faulty transformation is made if negative temperatures are computed incorrectly.

KMU (*Key-value Multiple Union*): A key and/or value that should be the union of several input attributes is incorrectly emitted. Consider a program that obtains the times that a temperature is repeated in a year. The *Map* function receives <year, temperature> and emits pairs <"yeartemperature", n>, where n is the number of times that a temperature appears in a year. Then the *Reduce* function aggregates all n values for each "yeartemperature" key. One fault appears when the *Map* function handles temperature and year as numbers, because if we have the year 1992 with 1º and the year 0199 with 21º, the *Map* function emits <19921, 1> and <19921, 1> respectively, which will produce a failure.

KMT (*Key-value Multiple Transforming*): A key and/or value that should be obtained after processing several input attributes is incorrectly emitted. Consider a program that calculates the average temperature of each year. The *Reduce* function receives pairs <year, list[temperatures]> and emits <year, average temperature>. *Reduce* function transforms list [temperatures] in "average temperature", and a failure occurs if it always returns the average in absolute value.

V. Functional testing with MRTree

This section shows how to derive test cases in order to detect the faults represented by MRTree in sub-section IV.

For each Fault-Node an example program is described. We provide guidelines on how to derive test cases related to the Fault-Node. Then we detail a possible fault that the program could have and the failure obtained after executing the test case. Finally we show how to execute the tests in order that different executions be repeatable.

URIO: Consider a program under test that obtains for each year the closest to zero negative temperature. The *Map* function receives years and temperatures in decreasing order and emits pairs <year, temperature>. The *Reduce* function receives pairs <year, list[temperatures]> and must repeat all temperatures returning for each year the closest to zero negative temperature.

The goal is to test the behavior in different situations determined by the order of the values at the input of *Reduce* function. A test case with year 2000 and temperatures 20º, 13º, 0º, -2º and -9º, should produce -2º for the year 2000.

A possible fault could be present if the *Reduce* function assumes that the values of temperatures arrive in descending order. The code with this fault is the following:

```
public void reduce (int key, Iterator
           <int> values){
    while (values.hasNext()){
        temp = values.next();
        if (temp < 0){
          emit(key, temp); break;
        }
    }
}
```

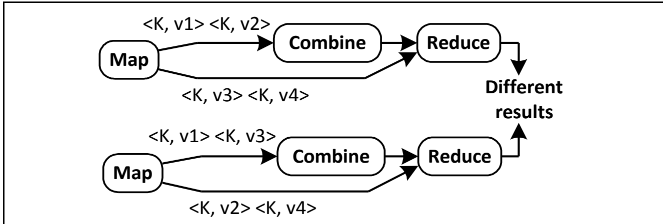

Fig. 3. Breaking the associativity between Combine and Reduce functions

With this *Reduce* function, the negative temperature closest to zero for the year 2000 is sometimes -2º (correct) and others -9º (incorrect). If the development environment is running the program without parallelism, the output will always be -2º.

To control the order of input values in the *Reduce* function, a test environment will be used with a single computer configured so that the input is implemented by the *Map* function without parallelism, so that the tests could be rerun with the same conditions.

UCSR: Consider a program under test that obtains for each year the average temperature. The *Map* function receives years and temperatures and emits pairs <year, temperature>. The *Reduce* function receives pairs <year, list[temperatures]> and returns the mean temperature of each year.

The goal is to test the behavior of the *Combine* function when running through different sets of pairs <key, value>. For the test case input, try to break the associativity between the *Combine* and *Reduce* functions. Given a key K, two values are emitted from *Map* to *Combine* and another two directly to *Combine*. The associativity is broken if a different result occurs when an input value from *Combine* and another from *Reduce* are exchanged. This situation is illustrated in Fig. 3 (note that values v2 and v3 have been exchanged).

The test case takes the years 1900 and 1901 as input, both with temperatures 1º, 2º, 3º and 4º; and *Combine* function receives as input <1900, [1º, 2º]> and <1901, [1º, 3º]>. The expected output is that the years 1900 and 1901 should have 2.5º as average temperature.

The program has a fault because the *Reduce* and *Combine* function, average function, is not associative, therefore the *Combine* function should not implement the average functionality. The output obtained is that the year 1900 has 2.83º as average temperature and 1901, 2.66º.

To run the tests we need an environment that allows us to control what key-value pairs are pre-processed by the *Combine* function before being sent to the *Reduce* function and what key-value pairs go directly to the *Reduce* function, as shown in Fig. 4.

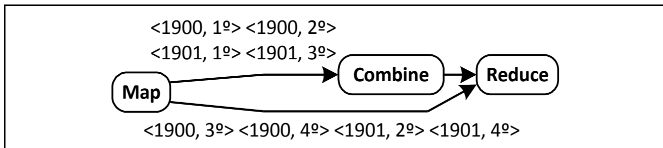

Fig. 4. Breaking the associativity between the Combine and Reduce functions in UCSR Fault-Node

UCDR: Consider a program under test that computes follow's asymmetric relations between Twitter contacts. Two persons, for example Harry and Lily have an asymmetrical relationship if Harry follows Lily but Lily doesn't follow Harry, or vice versa. The *Map* function receives an identifier of a person and obtains the follow's contacts, for example, it receives Harry follows Lily, and emits pairs <Harry, "Follow: Lily"> and <Lily, "Follower: Harry">. The *Reduce* function receives pairs <Person, list[Follows and followers]>, i.e., gets all the follows and all the followers of one person, so if a person is both follow and follower there is no asymmetric relationship; otherwise asymmetrical relationship exists. The implementation of *Reduce* is the following:

```
public void reduce (String key, Iterator
                 <String> values) {
   List<String> relationship = new
   ArrayList<String> (values);
   String contact;
   while (values.hasNext()){
      contact = values.next();
      if (isFollow (contact) &&
           !existFollower(contact,values)){
         emit(key, extractName(contact));
         emit(extractName(contact),key);
      }
   }
}
```

As in UCSR, the goal is to test the behavior when a *Combine* function runs over different sets of pairs <key, value> (Fig. 3). In this case *Combine* has a different implementation than *Reduce*.

If the program includes a *Combine* function to detect a symmetrical relationship, this kind of relationship is filtered and will not be sent to the *Reduce* function. The program has a fault because the *Combine* function can filter information that is apparently irrelevant in the subset of pairs <key, value> received, but will be necessary in the rest of the program pairs <key, value>, as will be seen later. The implementation of the *Combine* is the following:

```
public void combine (String key, Iterator
                         <String> values){
   List<String> relationship = new
   ArrayList<String> (values);
   String contact;
   while (values.hasNext()){
      contact = values.next();
      if ( (isFollow (contact) &&
              !existFollower (contact,values))
            || (isFollower (contact) &&
              !existFollow(contact,values)) ){
          emit (key, contact);
      }
   }
}
```

The test includes three users: @1, @2 and @3, where @1 and @3 follow and are followed by @2. The input of test case includes the user @1 three times, @3 another three times and then @2, where we also have to run a *Combine* with <@1, ["Follow: @2", "Follow: @2"]> y <@3, ["Follow: @2", "Follower: @2"]> as input. The expected output is that there are no asymmetric relationships between @1, @2 and @3.

The program causes a failure because the *Combine* function filters the pairs <@3, "Follow: @2"> and <@3, "Followed: @2"> and later this will be necessary for the correct execution of the *Reduce* function. The output obtained is that there are no asymmetrical relationship between @1 and @2, but there is an asymmetrical relationship between @3 and @2.

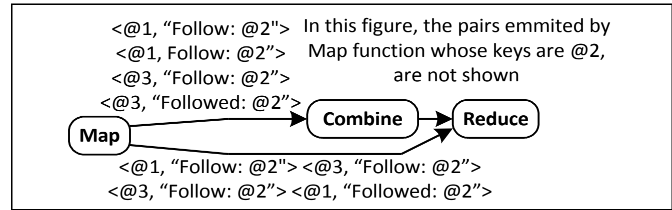

Fig. 5. Breaking the associativity between the Combine and Reduce functions in UCDR Fault-Node

The environment that allows testing in a controlled way should execute the test case as represented in Fig. 5.

CFNR: Consider a program under test that computes the maximum temperature of each year and calculates the average temperature of each 2-year range. Each range of 2 years is identified by the first year of the range, so that the *Map* function emits pairs <year, "year temperature"> and <year-1, "year temperature">.

The goal is to test the behavior of the program when the *Combine* function is not running and the <key, value> pairs are emitted directly from the *Map* function to the *Reduce* function. As in UCSR and in UCDR, for a K key emitted by *Map* function, we have to find v1, v2, v3 and v4 that satisfy Fig. 3.

Consider that *Combine* function computes the maximum temperature of each year, and the *Reduce* function, the average temperature. The inputs of test case are the years 2000 and 2004, both with temperatures 0º, 3º, 2º and 1º; where the *Combine* function will receive as input <2000 ["2000 3º", "2000 2º"]> and <2004 ["2004 2º", "2004 1º"]>. The expected output is that the ranges [1999-2000], [2000-2001], [2003-2004] and [2004-2005] have 3º as the average temperature of the maximum temperatures in each year for the range.

The program has a fault because the *Reduce* function does not implement the functionality performed in the *Combine* function. The *Combine* function calculates the maximum temperature of its input, but since it does not receive all temperatures for a year, the *Combine* function calculates a local maximum, while the *Reduce* function expects for each year the global maximum temperature. The output obtained is for the ranges [1999-2000] and [2003-2004], 1.5º; for the range [2000-2001], 1.33º; and for the range [2004-2005], 1.66º.

The environment that allows testing in a controlled way should execute the test case as represented in Fig. 6.

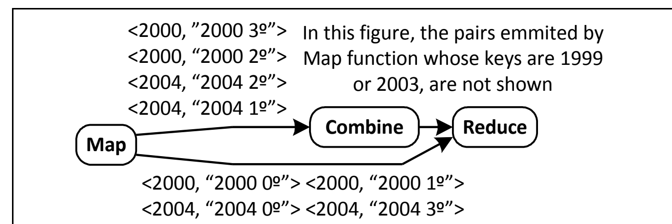

Fig. 6. Breaking the associativity between the Combine and Reduce functions in CFNR Fault-Node

VIT: Consider a program under test that obtains relationships of friendship and enmity on Twitter. The *Map* function receives two identifiers together with their tweets and emits <id_1, "Friend: id_2"> or <id_1, "Enemy: id_2">.

The goal is to test an inconsistency in the input of *Combine/Reduce* function. Inconsistency occurs when the input *Combine/Reduce* function has a pair <key, list[values]> where several values are incompatible with each other, and this situation makes the program cause a failure. Therefore, in the input for *Combine/Reduce* function a pair <K, [v1, v2]> is found to cause a failure in the program because v1 and v2 are inconsistent with each other.

For the above program, an inconsistency is that two persons are both friends and enemies in the input of the *Reduce* function. Separately, the *Map* function from one input determines that they are friends, but from the other input determines that they are enemies. The input of test case has a tweet from Jack and Jessica who declare that are friends, and another that indicates their enmity. The expected output is that Jessica is neither friend nor enemy of Jack.

If the *Reduce* function only stores the friends and enemies that it receives as input, then the program has a fault. With tweets for several years, two persons could be friends in the past and today enemies, so the *Reduce* function should consider this situation. In the faulty program, the actual output is that Jessica is friend and enemy of Jack.

KIT: Consider a program under test that obtains relationships of enmity on the basis of a series of tweets. The *Map* function receives tweets with several persons named, and for each person it emits a pair <person, tweet>. The *Reduce* function receives per each person all tweets where he/she is mentioned, these tweets are analyzed and the function obtains using all tweets which persons are enemies.

The goal is to test an inconsistency between a key and its value, i.e., independently, the contents of the key and the value are correctly calculated, but they cannot be together because they generate incompatibilities with each other. Thus an inconsistency between a key and its value is sought, i.e., both have been calculated correctly, but together they produce a special situation that the program does not expect and fails.

For the above program, an inconsistency is that the *Reduce* function emits a person that is the enemy of him/herself. The input of the test case is a tweet from William indicating self-loathing. The expected output is that William has no enemies.

If the *Reduce* function emits enemies regardless who people are, it has a fault. *Reduce* should not emit an enmity of a person with herself. In the faulty program the actual output is that William is enemy of himself.

VI. CONCLUSIONS

The MRTree classification allows us to identify potential faults that can occur in MapReduce programs under Hadoop. For the identified faults we provided a criterion for testing and illustrated it with some example programs. A number of examples have shown how tests could take into account this sort of faults specific to MapReduce.

As future work we plan to apply MRTree in industrial applications and formalize testing techniques applicable in Fault-Nodes.

ACKNOWLEDGMENT

This work has been performed under the research project TIN2013-46928-C3-1-R, funded by the Spanish Ministry of Economy and Competitiveness and ERDF Funds.

REFERENCES

[1] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In proc. 6th USENIX Sympnosium on Operating Systems Design and Implementation (OSDI), pages 137-149. USENIS, Dec. 2004.

[2] Open-source software for reliable, scalable, distributed computing http://hadoop.apache.org/

[3] T. White. Hadoop: The definitive guide. O'Reilly, 2012.

[4] Mahesh Gudipati, Shanthi Rao, Naju D. Mohan and Naveen Kumar Gajja. Big Data: Approach to Overcome Quality Challenges. In Big data: Challenges and opportunities. Infosys Labs Briefings. Vol 11 NO 1 2013.

[5] Java library that helps developers unit test Apache Hadoop map reduce jobs. http://mrunit.apache.org/.

[6] Shrinivas B. Joshi, Apache hadoop performance-tuning methodologies and best practices, Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, April 22-25, 2012, Boston, Massachusetts, USA.

[7] Kiyoung Kim, Kyungho Jeon, Hyuck Han, Shin G. Kim, Hyungsoo Jung, and Heon Y. Yeom. Mrbench: A benchmark for mapreduce framework. In *ICPADS '08: Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 11-18, Washington, DC, USA, 2008. IEEE Computer Society.

[8] Shengsheng Huang, Jie Huang, Yan Liu, Lan Yi and Jinquan Da et al. HiBench: A Representative and Comprehensive Hadoop Benchmark Suite.

[9] Benchmark for Hadoop clusters. http://hadoop.apache.org/docs/stable1/gridmix.html

[10] Development of packaging and tests of the Apache Hadoop ecosystem. http://bigtop.apache.org/

[11] J. Dorre, S. Apel, and C. Lengauer, "Static type checking of Hadoop MapReduce programs," in *Proceedings of the second international workshop on MapReduce and its applications.* ACM New York,USA, 2011, pp. 17–24.

[12] Christoph Csallner, Leonidas Fegaras y Chengkai Li. New Ideas Track: Testing MapReduce-Style Programs. Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. Pages 504-507.