ASTES

Advances in Science, Technology and Engineering Systems Journal Vol. 2, No. 1, 90-96 (2017)

www.astesj.com

Special Issue on Computer Systems, Information Technology, Electrical and Electronics Engineering

ASTESJ

ISSN: 2415-6698

Configuration/Infrastructure-aware testing of MapReduce programs

Jesús Morán[1]*, Bibiano Rivas[2], Claudio de la Riva[1], Javier Tuya[1], Ismael Caballero[2], Manuel Serrano[2]

[1]*University of Oviedo, Department of Computing, 33394, Spain*

[2]*University of Castilla-La Mancha, Institute of Technology and Information Systems, 13051, Spain*

ARTICLE INFO

Article history:
Received: 15 December, 2016
Accepted: 20 January, 2017
Online: 28 January, 2017

Keywords:
Software testing
Functional testing
MapReduce programs
Big Data Engineering
Hadoop

ABSTRACT

The implemented programs in the MapReduce processing model are focused in the analysis of large volume of data in a distributed and parallel architecture. This architecture is automatically managed by the framework, so the developer could be focused in the program functionality regardless of infrastructure failures or resource allocation. However, the infrastructure state can cause different parallel executions and some could mask the faults but others could derive in program failures that are difficult to reveal. During the testing phase the infrastructure is usually not considered because commonly the test cases contain few data, so it is not necessary to deploy a parallel execution or handle infrastructure failures, among others potential issues. This paper proposes a testing technique to generate and execute different infrastructure configurations given the test input data and the program under test. The testing technique is automatized by a test engine and is applied to real world case studies. As a result, the test engine generates and executes several infrastructure configurations, revealing a functional fault in two programs.

1. Introduction

The massive data processing trends have brought to light several technologies and processing models in the *Big Data Engineering* field [1]. Among them, *MapReduce* [2] can be highlighted as it permits the analysis of large data based on the "divide and conquer" principle. These programs run two phases in a distributed infrastructure: the *Mapper* and the *Reducer*. The first one divides the problem into several subproblems, and then the *Reducer* phase solves each subproblem. Usually, *MapReduce* programs run on several computers with heterogeneous resources and features. This complex infrastructure is managed by a framework, such as *Hadoop* [3] which stands out due to its wide use in the industry [4]. Other frameworks as for example Apache Spark [5] and Apache Flink [6] among others also use the *MapReduce* programming model.

From point of view of the developer, a *MapReduce* program can be implemented only with *Mapper* and *Reducer*, regardless of the infrastructure. Then the framework that manages the infrastructure is also responsible to, over several computers, automatically deploy, run the program and lead the data processing between the input and output. Among others, the framework divides the input into several subsets of data, then processes each one in parallel and re-runs some parts of the program if necessary.

Although that the program can be implemented abstracting the infrastructure, the developer needs to consider how the infrastructure configuration could affect the program functionality. A previous work [7] detects and classifies several faults that depend on how the infrastructure configuration affects the program execution and produces different output. These faults are often masked during the test execution because the tests usually run over an infrastructure configuration without considering the different situations that could occur in production, as for example different parallelism levels or the infrastructure failures [8]. On the other hand, if the tests are executed in an environment similar to the production, some faults may not be detected because it is common that the test inputs contain few data, and in these cases *Hadoop* does not parallelize the program execution. There are some tools to enable the simulation for some of these situations (for example computer and net failures) [9, 10, 11], but it is difficult to design, generate and execute the tests in a deterministic way because there are a lot of elements that need fine grained simulation, including the infrastructure and framework.

*Corresponding Author: Jesús Morán, Viesques edificio departamental 1 Gijón (Spain) moranjesus@lsi.uniovi.es

https://dx.doi.org/10.25046/aj020111

The main contribution of this paper is a technique that can be used to generate automatically the different infrastructure configurations for a *MapReduce* application. The goal is to execute test cases with these configurations in order to detect functional faults. Given a test input data, the configurations are obtained based on the different executions that can happen in production. Then each one of the configurations is executed in the test environment in order to detect functional faults of the program that may occur in production. This paper extends the previous work [12] and the contributions are:

1. A combinatorial technique to generate the different infrastructure configurations, taking into account characteristics related to the *MapReduce* processing and the test input data.
2. Automatic support by means of a test engine based on MRUnit [13] that allows the execution of the infrastructure configurations.
3. Evaluation of the approach detecting failures in a two real world programs.

The rest of the paper is organized as follows. In Section 2 the principles of the *MapReduce* paradigm are introduced. The related work about software testing in *MapReduce* paradigm is presented in Section 3. The generation of the different configurations, the execution and the automatization of the tests are defined in Section 4. In Section 5 it is applied to a two case studies. The paper ends with conclusions and future work in Section 6.

2. MapReduce Paradigm

The function of the *MapReduce* program is to process high quantities of data in a distributed infrastructure. The developer implements two functionalities: *Mapper* task that splits the problem into several subproblems and *Reducer* task that solves these subproblems. The final output is obtained from the deployment and the execution over a distributed infrastructure of several instances of *Mapper* and *Reducer*, also called tasks. Hadoop (or other framework) automatically carry out the deployment and execution. First, several *Mapper* tasks analyse in parallel a subset of input data and determine which subproblems these data need. When the execution of all *Mappers* are finished, several *Reducers* are also executed in parallel in order to solve the subproblems. Internally *MapReduce* handles <key, value> pairs, where the key is the subproblem identifier and the value contains the information to solve it.

To explain *MapReduce* let us suppose a program that calculates the average temperature per year from historical data about temperatures. This program solves for each year one subproblem, so the year is the identifier or key. The *Mapper* task receives a subset of temperature data and emits <year, temperature of this year> pairs. Then *Hadoop* aggregates all values per key. Therefore, the *Reducer* tasks receive subproblems like <year, [all temperatures of this year]>, that is all temperatures grouped per year. Finally, the *Reducer* calculates the average temperature. For example, in Figure. 1 an execution of the program considering the input is detailed: year 2000 with 3º, 2002 with 4º, 2000 with 1º, and 2001 with 5º. The first two inputs are analysed in one *Mapper* task and the remainder in another task. Then the temperatures are grouped per year and sent to the *Reducer* tasks. The first *Reducer* receives all the temperatures for the years 2000 and 2002, and the other task for the year 2001. Finally, each *Reducer* emits the average temperature of the analysed subproblems: 2º in the year 2000, 4º in 2002 and 5º in 2001. This program with the same input could be executed in another way by the framework, for example with three *Mappers* and three *Reducers*. Regardless of how the framework runs the program, it should generate the expected output.

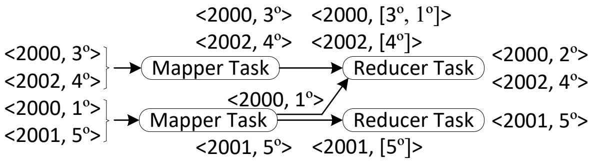

Figure. 1 Program that calculates the average temperature per year

In order to optimize the program, a *Combiner* functionality can be implemented. This task is run after the *Mapper* and the goal is to remove the irrelevant <key, value> pairs to solve the subproblem. In *MapReduce* there are also other implementations such as for example Partitioner that decides for each <key, value> pair which *Reducer* analyses it, Sort that sorts the <key, value> pairs, and Group that aggregates the values of each key before the *Reducer*.

An incorrect implementation of these functionalities could cause a failure in one of the different ways in which *Hadoop* can run the program. These faults are difficult to detect during testing because the test cases usually contain few input data. In this way it is not necessary to split the inputs and therefore the execution is over one *Mapper*, one *Combiner* and one *Reducer* [2].

3. Related Work

Despite the testing challenges of the *Big Data* applications [14, 15] and the progresses in the testing techniques [16], little effort is focused on testing the *MapReduce* programs [17], one of the principal paradigms of *Big Data* [18]. These large-scale programs have several issues and challenges to measure and assure the quality [19]. A study of Kavulya et al. [20] analyses several *MapReduce* programs and 3% of them do not finish, while another study by Ren et al. [21] places the number between 1.38% and 33.11%.

Many of the works about testing of the *MapReduce* programs focus on performance [22, 23, 24] and to a lesser degree functionality. A testing approach for *Big Data* is proposed by Gudipati et al. [25] specifying several processes, one of which is about *MapReduce* validation. In this process Camargo et al. [26] and Morán et al. [7] identify and classify several functional faults. Some of these faults are specific of the *MapReduce* paradigm and they are not easy to detect because they depend on the program execution over the infrastructure. One common type of fault is produced when the data should reach the *Reducer* in a specific order, but the parallel execution causes these data to arrive disordered. This fault was analysed by Csallner et al. [27] and Chen et al. [28] using some testing techniques based on symbolic execution and model checking. In contrast to the previous works, the approach of this paper is not focused on the detection of only one type of fault, it can also detect other *MapReduce* specific faults. To do this, the test input data is executed over different infrastructure configurations that could lead to failures.

Several research lines suggest injecting infrastructure failures [29, 30] during the testing, and several tools support their injection [9, 10, 11]. For example, the work by Marynowski et al. [31] allows the creation of test cases specifying which computers fail and when. One possible problem is that some specific *MapReduce* faults could not be detected by infrastructure failures, but require full control of *Hadoop* and the infrastructure. In this paper, the different ways in which *Hadoop* could run the program are automatically generated from the functional point of view, regardless of the infrastructure failures and *Hadoop* optimizations.

Furthermore, there are other approaches oriented to obtain the test input data of *MapReduce* programs, such as [32] that employs data flow testing, other based on a bacteriological algorithm [33], and [34] based on input domain together with combinatorial testing focused on ETL (Extract, Transformation and Load). In this paper, given a test input data, several configurations of infrastructure are generated and then executed in order to reveal functional faults. The test input data of this approach could be obtained with the previous testing techniques.

The functional tests can be executed directly in the production cluster or in one computer with *Hadoop*. Herriot [35] can be used to execute the tests in a cluster while providing access to their components supporting, among others, the injection of faults. Another option is to simulate a cluster in memory with the MiniClusters libraries [36]. In the unit testing, JUnit [37] could be used together with mock tools [38], or directly by MRUnit library [13] adapted to the *MapReduce* paradigm. These test engines only execute one infrastructure configuration and usually without parallelization. In this paper a test engine is implemented by an MRUnit extension that automatically generates and executes the different infrastructure configurations that could occur in production. The test engine proposed extends MRUnit because in Hadoop is very usual to develop java programs [21], and the java programs usually employs JUnit libraries [39]. MRUnit put together JUnit with mocks, reflection and other tools in order to simplify the execution of the test case for the *MapReduce* programs.

4. Generation and Execution of Tests

The generation of the infrastructure configurations for the tests are defined in Section 4.1, and a framework to execute the tests in Section 4.2.

4.1. Generation of the test scenarios

To illustrate how the infrastructure configuration affects the program output, suppose that the example of Section 2 is extended with a *Combiner* in order to decrease the data and improve the performance. The *Combiner* receives several temperatures and then they are replaced by their average in the *Combiner* output. This program does not admit a *Combiner* because all the temperatures are needed to obtain the total average temperature. The *Combiner* is added in order to optimize the program, but injects a functional fault in the program. Figure. 2 represents three possible executions of this program with the same input (year 1999 with temperatures 4º, 2º and 3º) that could happen in production considering the different infrastructure configurations.

The first configuration executes one *Mapper*, one *Combiner* and one *Reducer* and produces the expected output. The second configuration also generates the expected output executing one *Mapper* that processes the temperatures 4º and 2º, another *Mapper* for 3º, two *Combiner*, and finally one *Reducer*. The third configuration also executes two *Mapper*, two *Combiner* and one *Reducer*, but produces an unexpected output because the first *Mapper* processes 4º and the second *Mapper* the temperatures 2º and 3º. Then one of the *Combiner* tasks calculates the average of 4º, and the other *Combiner* of 2º and 3º. The *Reducer* receives the previous averages (4º and 2.5º), and calculates the total average in the year. This configuration produces 3.25º as output instead of the 3º of the expected output. The program has a functional fault only detected in the third configuration. Whenever this infrastructure configuration is executed the failure is produced, regardless of the computer failures, slow net or others. This fault is difficult to reveal because the test case needs to be executed in a completely controlled way under the infrastructure configuration that detect it.

Given a test input data, the goal is to generate the different infrastructure configurations, also called in this context *scenarios*. For this purpose, the technique proposed considers how the *MapReduce* program can execute these input data in production. First, the program runs the *Mappers*, then over their outputs the *Combiners* and finally the *Reducers*. The execution can be carried out over a different number of computers and therefore the *Mapper*-*Combiner*-*Reducer* can analyse a different subset of data in each execution. In order to generate each one of the *scenarios*, a combinatorial technique [40] is proposed to combine the values of the different parameters that can modify the execution of the *MapReduce* program. In this work the following parameters are considered based on previous work [7] that classifies different types of faults of the *MapReduce* applications:

- *Mapper* parameters: (1) Number of *Mapper* tasks, (2) Inputs processed per each *Mapper*, and (3) Data processing order of the inputs, that is, which data are processed before other data in the *Mapper* and which data are processed after.
- *Combiner* parameters for each *Mapper* output: (1) Number of *Combiner* tasks, and (2) Inputs processed per each *Combiner*.
- *Reducer* parameters: (1) Number of *Reducer* tasks, and (2) Inputs processed per each *Reducer*.

The different *scenarios* are obtained through the combination of all values that can take the above parameters and applying the constraints imposed by the sequential execution of *MapReduce*. The constraints considered in this paper are the following:

1. The values/combinations of the *Mapper* parameters depend on the input data because it is not possible more tasks than data. For example, if there are three data items in the input, the maximum number of *Mappers* is three.
2. The values/combinations of the *Combiner* parameters depend on the output of the *Mapper* tasks.
3. The values/combinations of the *Reducer* parameters depend on the output of the *Mapper*-*Combiner* tasks and another functionality executed by *Hadoop* before *Reducer* tasks. This other functionality is called Shuffle and for each <key, value> pair determines the *Reducer* task that requires these data, then sorts all the data and aggregates by key.

Suppose the program of Figure. 2 to illustrate how the parameters are combined and how the constraints are applied. The input of this program contains three records, and these data constrain the values that the *Mapper* parameters can take because the maximum number of *Mapper* tasks is three (one *Mapper* per each <key, value> pair). The first *scenario* is generated with one *Mapper*, one *Combiner* and one *Reducer*. For the second *scenario*

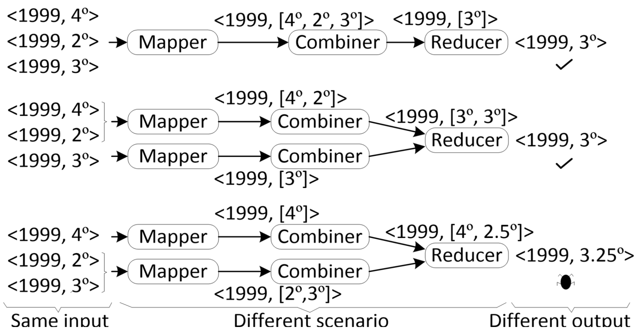

Figure. 3 Different infrastructure configurations for a program that calculates the average temperature per year with Combiner task

the parameter "Number of *Mapper* tasks" is modified to 2, where the first *Mapper* analyses two <key, value> pairs, and the second processes one pair. The third *scenario* maintains the parameter "Number of *Mapper* tasks" at 2, but modifies the parameter "Inputs processed per each *Mapper*", so the first *Mapper* analyses one <key, value> pair and the other *Mapper* processes two pairs. The *scenarios* are generated by the modification of the values in the parameters in this way and considering the constraints

4.2. Execution of the test scenarios

The testing technique proposed in the previous section is focused in the generation of *scenarios* that represent different infrastructure configurations according to the characteristics of the *MapReduce* processing. The test cases are systematically executed in these scenarios according to the framework described in Figure. 2.

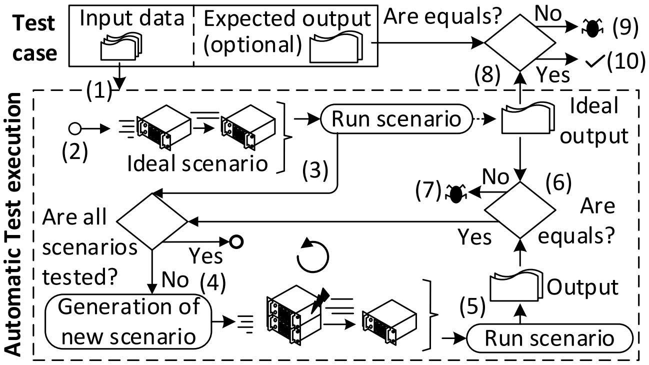

Figure. 2 General famework of test execution

The framework takes as input a test case that contains the input data and optionally the expected output. The test input data can be obtained with a generic testing technique or one specifically designed for *MapReduce*, such as MRFlow [32]. Then, the *ideal scenario* is generated (1) and executed (2, 3). This is the *scenario* formed by one *Mapper*, one *Combiner* and one *Reducer* which is the usual configuration executed in testing. Next, new *scenarios* are iteratively generated (4) and executed (5) through the technique of the previous section. The output of each *scenario* is checked against the output of the *ideal scenario* (6), revealing a fault if the outputs are not equivalent (7). Finally, if the test case contains the expected output, the output of *ideal scenario* is also checked against the expected output (8), detecting a fault when both are not equivalent (9, 10).

Given a test case, the *scenarios* are generated according to the previous section, then they are iteratively executed and evaluated following the following pseudocode:

```
Input: Test case with:
   input data
   expected output (optional)
Output: scenario that reveals a fault
     /* Generation of scenarios (section 4.1)*/
     Scenarios ← Generate scenarios from input data
     /* Execution of scenarios */
     ideal scenario output ← Execution of ideal scenario
     ∀ scenario ∈ Scenarios:
       scenario output ← Execution of scenario
       IF scenario output <> ideal scenario output:
         RETURN scenario with fault
     IF ideal scenario output <> expected output:
       RETURN ideal scenario
     ELSE:
       RETURN Zero faults detected
```

For example, Figure. 2 contains the generation and execution of a program that calculates the average temperature per year in three *scenarios* considering the same test input: year 1999 with temperatures 4º, 2º and 3º. The first execution is the *ideal scenario* that produces 3º as output through one *Mapper*, one *Combiner* and one *Reducer*. Then the second *scenario* that contains two Mappers and two Combiners is executed and also produces 3º. Finally, a third *scenario* with two *Mappers* and two *Combiners* is executed, but with different information in the *Mappers* than the second *scenario*, and produces 3.25º as output. This temperature is not equivalent to the 3º of the *ideal scenario* output. Consequently, a functional fault is revealed without any knowledge of the expected output of the test case.

This approach is automatized by means of a test engine based on MRUnit library [13]. This library is used to support the execution of each *scenario*. In MRUnit the test cases are executed in the *ideal scenario*, but this library is extended to generate other *scenarios* and enable parallelism. In order to support the execution of several *Mapper*, *Combiner* and *Reducer* tasks, MRUnit is extended providing support for advanced functionalities as for example customized *Partitioners*.

5. Case Studies

The following two real world programs are used as case studies in order to evaluate the proposed approach: (1) the Open Ankus recommendation system [41], and (2) the *MapReduce* program described in I8K|DQ-BigData framework [42]. Each case study is detailed in the below sections.

5.1. Open Ankus recommendation system

This recommendation system is part of a machine learning library implemented in the *MapReduce* paradigm. The system predicts and recommends several items (books, films or others) to each user based on the personal tastes saved in the profile. One functionality checks the accuracy of the recommendations based on the points predicted by the systems against points assigned by the users for each item. This functionality has a MultipeInputs [43] design that consists in two different *Mapper* implementations: one receives the points predicted by the system and the other the points assigned by the user, but both *Mappers* emit data to the same *Reducer* implementation. The *Mappers* tasks receive from all users and all items the points predicted and the points assigned, and then the *Mapper* aggregates these points for each user-item pair. The

Reducer tasks receive for each user-item all points predicted by the system and all points assigned by the user, and then calculate the accuracy of the predictions.

For this program a test case is obtained using the MRFlow testing technique based on data flow adaptation to the *MapReduce* programs [7]. The test input data contain two predictions and two user assignments for one item: (1) the system predicts that Carol could assign 0 points to Don Quixote item, (2) Carol assigns 0 points to Don Quixote, (3) later the system detects a change in the Carol taste and predicts that Carol could assign 10 points to Don Quixote, and (4) Carol assigns 10 points to Don Quixote. These data are passed to the test engine saved in two files, one for predictions and other for assignments. The expected output is 100% of accuracy in the predictions.

The procedure described in Section 4 is applied on the previous program using the previous test case as input. As a result, a fault is detected and causes a failure when some inputs are processed before others. In this case, the program should check the points assigned by the user against their predictions, but could check it against other predictions and then a wrong accuracy could be obtained. In the previous test case, a failure occurs when the system checks the prediction 0 against the 10 points assigned by the user instead to check against the 0 points assigned by the user. The bottom of Figure. 4 represents one *scenario* that reveals the failure. This *scenario* starts with one *Mapper* to analyse the predictions and other two *Mappers* for the points assigned, 0 and 10 respectively. In this *scenario* the 10 points assigned by the user are analysed before the 0 points also assigned by the user. The *Reducer* task receives several points predicted and assigned by the user, and then checks the first prediction against the first points assigned by the user, and so on. In this *scenario* the *Reducer* task receives: (1) the predictions 0 and 10 points, and (2) the points assigned by the user, 10 and 0. This *Reducer* task generates a wrong accuracy because checks the 0 points predicted against the 10 points assigned by the user instead the 0 points. As output, the system did not predict well, but given the test input data the system should have predicted perfectly.

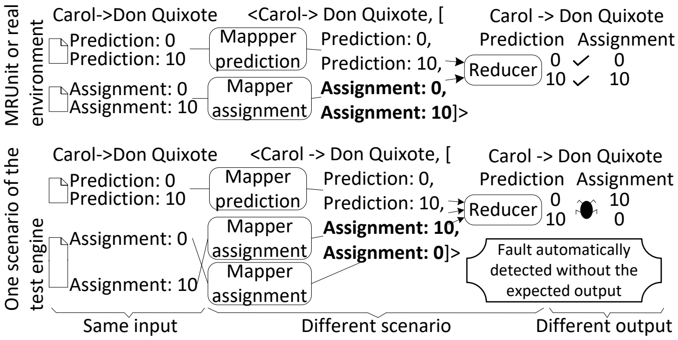

Figure. 4 Execution of the Open Ankus test case in different scenarios

The testing technique proposed in this paper detects the fault with the previous test case. However, the following test environments do not detect the fault: (a) *Hadoop* cluster in production with 4 computers, *Hadoop* in local mode (simple version of *Hadoop* with one computer), and (c) MRUnit unit testing library. These environments mask the fault because only execute the test cases in one *scenario* represented in the top of Figure. 4. This *scenario* is the *ideal scenario* with only two *Mappers* due a MultipleInputs design of the program: one for predictions and other for points assigned by user. In the *ideal scenario* there is no parallelization for predictions and points assigned by user, then the fault is masked.

The test engine proposed in this paper detects the fault because executes the test case in several *scenarios* that could happen in production. In contrast with the other environments, this test engine does not need the expected output to reveal the fault. At first point, the test engine obtains the output from the *ideal scenario* and then checks if the other *scenario* produces an equivalent output or not. For example, in the previous test case, for the same input some *scenarios* produce one output (the system predictions are perfect) and other *scenarios* produce different output (the system predictions are wrong).

5.2. 18K|DQ-BigData framework

This program measures the quality of the data exchanged between organizations according to part 140 of the ISO/TS 8000 [44]. The program receives (1) the data exchanged in a row-column fashion, together with (2) a set of mandatory columns that should contain data and (3) a percentage threshold that divides the data quality of each row in two parts: the first part is maximum if all mandatory columns contain data and zero otherwise, and the second part of the data quality is calculated as the percentage of the non-mandatory columns that contain data. The output of the program is the data quality of each row, and the average of all rows.

Over the previous program, a test case is obtained using again a specific *MapReduce* testing technique based on data flow [7]. The test input data and the expected output of the test case contain two rows represented in Table. 1. Row 1 contains two columns (Name and City), and only one column has data, so the data quality is 50%. Row 2 contains data in all columns, so the data quality is 100%. The total quality is 75%, which is the average of both rows.

Input		**Expected output**	
Data quality threshold: 50%			
Mandatory columns: "Name"			
Row 1	Name: Alice	50%	75% (Average)
	City: (no data)		
Row 2	Name: Bob	100%	
	City: Vienna		

Table. 1 TEST CASE OF THE I8K|DQ-BIGDATA PROGRAM

The procedure described in Section 4 is applied on the previous program using the previous test case as input. As a result, a fault is detected and reported to the developer. This failure occurs when the rows are processed in different *Mappers* and only the first *Mapper* receives the information related to the mandatory columns and the data quality threshold, because *Hadoop* splits the input data into several subsets. Without this information, the *Mapper* cannot calculate the data quality and does not emit any output. The bottom of Figure. 5 represents the *scenario* that produces the failure. There are two *Mappers* that process different rows. The first *Mapper* receives the data quality threshold (value of 50%), the mandatory column ("Name") and the two columns of row 1 with only data in one column, so the *Mapper* emits 50% as data quality of row 1. The second *Mapper* processes only row 2, but no other information about the mandatory columns or data quality threshold, so this *Mapper* cannot emit any output. Then the *Reducer* receives only the data quality of row 1 and emits an incorrect output of the average data quality.

This fault is difficult to detect because it implies the parallel and controlled execution of the program. Moreover, this fault is not revealed by the execution of the test case in the following environments: (a) *Hadoop* cluster in production with 4 computers, *Hadoop* in local mode (simple version of *Hadoop* with one computer), and (c) MRUnit unit testing library. These environments do not detect the fault because they only execute one *scenario* that masks the fault. Normally these environments run the program in the *ideal scenario* that is formed by one *Mapper*, one *Combiner* and one *Reducer*, and then the fault is masked due to a lack of parallelism.

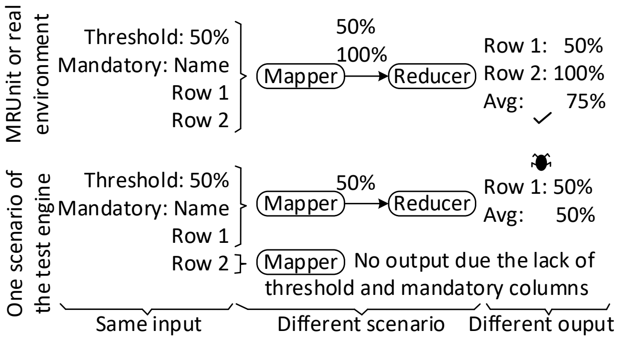

Figure. 5 Execution of the 18K|DQ-BigData test case in different scenarios

The test engine proposed in this paper executes the test case in the different *scenarios* that can occur in production with large data and infrastructure failures. In contrast with the other environments, the test engine proposed does not need the expected output to detect faults. For example, in this case study the fault is revealed automatically because the outputs of the different *scenarios* are not equivalent to each other. The execution of some *scenarios* obtains an average quality of 75%, whereas the execution of other *scenarios* obtains 50%. These outputs are not equivalent, and the test engine detects automatically a fault despite the unknown expected output.

After the detection and report of the fault during the test phase, the developer fixed the program and then the test case passed.

6. Conclusions

A testing technique for *MapReduce* applications is described and automatized as a test engine that generates and executes different infrastructure configurations for a given test case. This test engine can detect automatically functional faults related to the *MapReduce* paradigm without the expected output. In general, these design faults are difficult to detect in test/production environments because the execution is performed without parallelization or infrastructure failures. This testing technique is applied in test cases with little data in two real world programs. As a result, functional faults are revealed automatically.

As future work the generation of the infrastructure configurations could be improved by the extension of the testing technique in order to select efficiently the configurations that are more likely to detect faults. The current approach is *off-line* because the tests are not carried out when the program is in production. As future work we plan to extend the approach to *on-line* testing, in order to monitor the functionality with the real data when the program is executed in production and detect the faults automatically.

Conflict of Interest

The authors declare no conflict of interest.

Acknowledgment

This work was supported in part by PERTEST (TIN2013-46928-C3-1-R), project funded by the Spanish Ministry of Science and Technology; TESTEAMOS (TIN2016-76956-C3-1-R) and SEQUOIA (TIN2015-63502-C3-1-R), projects funded by the Spanish Ministry of Economy and Competitiveness; GRUPIN14-007, funded by the Principality of Asturias (Spain); CIEN LPS-BIGGER project; and ERDF funds.

References

[1] ISO/IEC JTC 1 – Big Data, preliminary report 2014, ISO/IEC Std., 2015.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in Proc. of the OSDI - Symp. on Operating Systems Design and Implementation. USENIX, 2004, pp. 137–149.

[3] Apache Hadoop: open-source software for reliable, scalable, distributed computing, https://hadoop.apache.org, accessed: 2017-01-16.

[4] Institutions that are using Apache Hadoop for educational or production uses, http://wiki.apache.org/hadoop/PoweredBy, accessed: 2017-01-16.

[5] Apache Spark: a fast and general engine for large-scale data processing, https://spark.apache.org, accessed: 2017-01-16.

[6] Apache Flink: Scalable batch and stream data processing, https://-flink.apache.org/, accessed: 2017-01-16.

[7] J. Morán, C. de la Riva, and J. Tuya, "MRTree: Functional Testing Based on MapReduce's Execution Behaviour," in Future Internet of Things and Cloud (FiCloud), 2014 International Conference on, 2014, pp. 379–384.

[8] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in Proceedings of the 1st ACM symposium on Cloud computing. ACM, 2010, pp. 193–204.

[9] AnarchyApe: Fault injection tool for hadoop cluster from yahoo anarchyape, https://github.com/david78k/anarchyape, accessed: 2017-01-16.

[10] Chaos Monkey, https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey, accessed: 2017-01-16.

[11] Hadoop injection framework, https://hadoop.apache.org, accessed: 2017-01-16.

[12] J. Moran, B. Rivas, C. De La Riva, J. Tuya, I. Caballero, and M. Serrano, "Infrastructure-aware functional testing of mapreduce programs," 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW), Aug 2016. [Online]. Available: http://dx.doi.org/-10.1109/W-FiCloud.2016.45

[13] Apache MRUnit: Java library that helps developers unit test Apache Hadoop map reduce jobs, http://mrunit.apache.org, accessed: 2017-01-16.

[14] S. Nachiyappan and S. Justus, "Getting ready for bigdata testing: A practitioner's perception," in Computing, Communications and Networking Technologies (ICCCNT), 2013 Fourth International Conference on. IEEE, 2013, pp. 1–5.

[15] A. Mittal, "Trustworthiness of big data," International Journal of Computer Applications, vol. 80, no. 9, 2013.

[16] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in Future of Software Engineering, 2007. FOSE '07, 2007, pp. 85–103.

[17] L. C. Camargo and S. R. Vergilio, "Mapreduce program testing: a systematic mapping study," in Chilean Computer Science Society (SCCC), 32nd International Conference of the Computation, 2013.

[18] M. Sharma, N. Hasteer, A. Tuli, and A. Bansal, "Investigating the inclinations of research and practices in hadoop: A systematic review," confluence The Next Generation Information Technology Summit (Confluence), 2014 5th International Conference -.

[19] J. Merino, I. Caballero, B. Rivas, M. Serrano, and M. Piattini, “A data quality in use model for big data,” Future Generation Computer Systems, 2015.

[20] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, “An analysis of traces from a production mapreduce cluster,” in Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on. IEEE, 2010, pp. 94–103.

[21] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, “Hadoop’s adolescence: an analysis of hadoop usage in scientific workloads,” Proceedings of the VLDB Endowment, vol. 6, no. 10, pp. 853–864, 2013.

[22] M. Ishii, J. Han, and H. Makino, “Design and performance evaluation for hadoop clusters on virtualized environment,” in Information Networking (ICOIN), 2013 International Conference on, 2013, pp. 244–249. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6496384

[23] Z. Liu, “Research of performance test technology for big data applications,” in Information and Automation (ICIA), 2014 IEEE International Conference on. IEEE, 2014, pp. 53–58.

[24] G. Song, Z. Meng, F. Huet, F. Magoules, L. Yu, and X. Lin, “A hadoop mapreduce performance prediction method,” in High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on, 2013, pp. 820–825. [Online]. Available: http://-ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6832000

[25] M. Gudipati, S. Rao, N. D. Mohan, and N. K. Gajja, “Big data: Testing approach to overcome quality challenges,” Big Data: Challenges and Opportunities, pp. 65–72, 2013.

[26] L. C. Camargo and S. R. Vergilio, “Cassicação de defeitos para programas mapreduce: resultados de um estudo empírico,” in SAST - 7th Brazilian Workshop on Systematic and Automated Software Testing, 2013.

[27] C. Csallner, L. Fegaras, and C. Li, “New ideas track: testing mapreduce-style programs,” in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, 2011, pp. 504–507.

[28] Y.-F. Chen, C.-D. Hong, N. Sinha, and B.-Y. Wang, “Commutativity of reducers,” in Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2015, pp. 131–146.

[29] F. Faghri, S. Bazarbayev, M. Overholt, R. Farivar, R. H. Campbell, and W. H. Sanders, “Failure scenario as a service (fsaas) for hadoop clusters,” in Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management. ACM, 2012, p. 5.

[30] P. Joshi, H. S. Gunawi, and K. Sen, “Prefail: A programmable tool for multiple-failure injection,” in ACM SIGPLAN Notices, vol. 46, no. 10. ACM, 2011, pp. 171–188.

[31] J. E. Marynowski, A. O. Santin, and A. R. Pimentel, “Method for testing the fault tolerance of mapreduce frameworks,” Computer Networks, vol. 86, pp. 1–13, 2015.

[32] J. Morán, C. de la Riva, and J. Tuya, “Testing Data Transformations in MapReduce Programs,” in Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation, ser. A-TEST 2015. New York, NY, USA: ACM, 2015, pp. 20–25.

[33] A. J. Mattos, “Test data generation for testing mapreduce systems,” in Master’s degree dissertation, 2011.

[34] N. Li, Y. Lei, H. R. Khan, J. Liu, and Y. Guo, “Applying combinatorial test data generation to big data applications,” Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016, 2016. [Online]. Available: http://dx.doi.org/10.1145/2970276.2970325

[35] Herriot: Large-scale automated test framework, https://wiki.apache.org/-hadoop/HowToUseSystemTestFramework, accessed: 2017-01-16.

[36] Minicluster: Apache Hadoop cluster in memory for testing, https://-hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/-CLIMiniCluster.html, accessed: 2017-01-16.

[37] JUnit: a simple framework to write repeatable tests, http://junit.org/, accessed: 2017-01-16.

[38] Mockito: Tasty mocking framework for unit tests in java, http://mockito.org/, accessed: 2017-01-16.

[39] D. Qiu, B. Li, and H. Leung, “Understanding the api usage in java,” Information and Software Technology, vol. 73, pp. 81–100, 2016.

[40] M. Grindal, J. Offutt, and S. F. Andler, “Combination testing strategies: a survey,” Software Testing, Verification and Reliability, vol. 15, no. 3, pp. 167–199, 2005.

[41] Open Ankus: Data mining and machine learning based on mapreduce, http://-www.openankus.org/, accessed: 2017-01-16.

[42] B. Rivas, J. Merino, M. Serrano, I. Caballero, and M. Piattini, “I8k| dq-bigdata: I8k architecture extension for data quality in big data,” in Advances in Conceptual Modeling. Springer, 2015, pp. 164–172.

[43] Multipleinputs: library to support mapreduce jobs that have multiple input paths with a different inputformat and mapper for each path, https://-hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/lib/-MultipleInputs.html, accessed: 2017-01-16.

[44] ISO/TS 8000-140, Data quality - Part 140: Master data: Exchange of characteristic data: Completeness, ISO/TS Std., 2009.