Testing Data Transformations in MapReduce Programs

Jesús Morán
University of Oviedo
Computer Science Department
Campus de Viesques, Gijón, Spain
(+34) 985 18 2153
moranjesus@lsi.uniovi.es

Claudio de la Riva
University of Oviedo
Computer Science Department
Campus de Viesques, Gijón, Spain
(+34) 985 18 2664
claudio@uniovi.es

Javier Tuya
University of Oviedo
Computer Science Department
Campus de Viesques, Gijón, Spain
(+34) 985 18 2049
tuya@uniovi.es

ABSTRACT
MapReduce is a parallel data processing paradigm oriented to process large volumes of information in data-intensive applications, such as Big Data environments. A characteristic of these applications is that they can have different data sources and data formats. For these reasons, the inputs could contain some poor quality data that could produce a failure if the program functionality does not handle properly the variety of input data. The output of these programs is obtained from a number of input transformations that represent the program logic. This paper proposes the testing technique called MRFlow that is based on data flow test criteria and oriented to transformations analysis between the input and the output in order to detect defects in MapReduce programs. MRFlow is applied over some MapReduce programs and detects several defects.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification – *Validation*

General Terms
Reliability, Verification.

Keywords
Software Testing, Data Flow Testing, MapReduce programs.

1. INTRODUCTION
The *MapReduce* paradigm [11] is based on the "divide and conquer" principle, which is the breaking down (*Map*) of a large problem into several sub-problems (*Reduce*). *MapReduce* is used in *Big Data* and *Cloud Computing* to process large data. The unit of program information is a <*key, value*> pair, where the *value* has data relative to the sub-problem identified by the *key*. The program output is the result of a series of transformations about the input information stored in the <*key, value*> pairs.

The quality in *MapReduce* programs is important due to their use in critical sectors, like health (ADN alignment [27]) or security (image processing in ballistics [17]). Software testing is one of the industrial practices most used to ensure quality. In recent years testing technique research has advanced [6], but few efforts have been focused on massive data processing like *MapReduce* [8]. These paradigms have new challenges in the field of testing [23][21][29], and some authors [15][26] estimate respectively that 3% and 1.38%-33.11% of *MapReduce* programs do not finish. Another *MapReduce* issue is that in some scenarios the developers create several subprograms with a few transformations instead of creating one program [26]. In these scenarios, the subprograms take more resources and underperform in comparison with a whole program.

On the other hand, a study about the *MapReduce* field has discovered that 84.5% of faults are due to data processing [19]. In order to detect these defects, this paper proposes a testing technique that analyzes the program transformations which could produce the failures. The testing technique named *MRFlow* (MapReduce data Flow) is based on *data flow* test criteria [25]. The program functionality is represented by means of program transformations, and then the test cases are derived from these transformations in order to test the functionality. Firstly, a program graph is elaborated with information about the program transformations, then the paths under test are extracted representing the transformations, and finally each path under test is tested with different data (empty, not empty, valid, non-valid, with emission of result and without emission of result). The main contributions of this paper are (1) a testing technique specifically tailored to test *MapReduce* programs in order to detect defects, and (2) the application over two popular case studies.

The rest of the paper is organized as follows: the *MapReduce* paradigm, *data flow* test criteria and the related work are summarized in Section 2. Next, Section 3 describes the *MRFlow* testing technique, the elaboration of the graph in Subsection 3.1 and the derivation of test cases in Subsection 3.2. In Section 4 *MRFlow* is applied to two programs and reveals some defects. Finally, Section 5 contains the conclusions.

2. BACKGROUND
The *MRFlow* testing technique is based on *data flow* criteria that analyze the evolution of variables in *MapReduce* programs. In Subsection 2.1 the *MapReduce* paradigm is summarized, *data flow* test criteria basis is in Subsection 2.2, and the related work is described in Subsection 2.3.

2.1 MapReduce
The *MapReduce* paradigm solves a problem by splitting it into sub-problems that can run in parallel. Fundamentally, *MapReduce* has two functions: *Map* that splits the problem into sub-problems, and *Reduce* which solves each sub-problem. Both functions handle <*key, value*> pairs, where *key* is the identifier of each sub-

A-TEST'15, August 30-31, 2015, Bergamo, Italy

http://dx.doi.org/10.1145/2804322.2804326

problem and the *value* corresponds to some data relative to that sub-problem. The *Map* function receives the data input and emits a <*key, value*> pair, then the *Reduce* function receives <*key, list(values)*> pairs that contain all the information about each sub-problem, and finally solves it with a <*key, value*> pairs.

Consider as an example a program that counts the number of occurrences of each word in a text. This problem is divided into as many sub-problems as there are different words, then each sub-problem only counts the occurrences of one word and the *key* is that word. The goal of the program is to count, so the *value* should contain information relative to the counting of the word, then the *value* contains a number of occurrences. For example, if the input texts are "hi Hadoop" and "hi", the *Map* function emits <*hi, 1*>, <*Hadoop, 1*> and <*hi, 1*>. Then there are two sub-problems, so the *Reduce* function receives <*Hadoop, 1*> and <*hi, [1,1]*> and emits <*Hadoop, 1*>, <*hi, 2*> which is the number of occurrences of each word in the texts.

The *MapReduce* programs are often used in *Big Data* programs [28], which process large data (Volume), with a necessary performance (Velocity) and with different types of data, data from different sources, and data without apparently a data model such as for example emails or videos (Variety). To handle this data a parallel and fault tolerant infrastructure is necessary, for this reason typically the *MapReduce* programs run over frameworks, excelling *Hadoop* [1] due to its impact on corporations [2].

2.2 Data Flow Test Criteria

The goal of *data flow* test criteria is to derive tests through the analysis of program variables. Several testing techniques are based on *data flow*, for example to test web applications through the analysis of state variables [5]. *Data flow* is a structure testing technique [4] created from the program *P*. A control flow graph *G(P)* is created from the program, where the edges represent each statement, and the vertices indicate the following possible statements. In addition to the graph, the definition and uses of every variable are determined [25]. In a node $n \in N$, when a *value* is assigned to the variable $v \in V$, the variable *v* is defined and the representation is *DEF(v,n)*. If the variable *v* is in a predicate of a condition (i.e., if (v)), then the representation is *P-USE(v,n)*, and in other uses of *v* the representation is *C-USE(v,n)*. For example, in the statement a = b+1, *a* is defined and *b* is used.

2.3 Related Work

Several testing approaches exist over the *MapReduce* programs, but most of them are focused on testing the performance [16][14][9] and few are oriented to testing the functionality, that is the goal of this paper. A classification of testing in *Big Data* is proposed by Gudipati et al. [13]. On this point Camargo et al. [7] and Morán et al. [22] elaborate a classification of defects, and Csallner et al. [10] test one defect automatically based on a symbolic execution framework. Another defect can be detected in compilation time by Dörre et al. [12]. In order to create test inputs, Mattos [20] develops a bacteriological algorithm supported by a function created by the tester, and Li et al. [18] design a test framework which validates the large database procedures. Our paper is different from other studies in the sense that it obtains the test cases from the program transformations systematically.

3. MRFLOW TESTING TECHNIQUE

The *MapReduce* program logic is represented by the transformations of *keys* and *values* into the program output. In these transformations, the *keys* and *values* can be transformed into one variable, this variable can be transformed into another, and so on until the final output.

Usual *data flow* test criteria like "*all-du-paths*" analyzes the definitions and uses of each variable, but does not consider the transformations between variables in enough degree of detail. In this sense, the testing technique proposed (*MRFlow*) analyzes the transformations from *keys* and *values*. This paper focuses on the *Reduce* function because it has a large part of the program functionality, but it can also be applied over the *Map* function because both handle *key* and *values*. Subsection 3.1 describes the elaboration of the graph, and the derivation of the test is detailed in Subsection 3.2.

3.1 Elaboration of MRFlow Graph

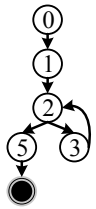
In the *MRFlow* graph, the statements of the program are in the nodes and each edge represents the next potential statement. In this graph, as described below, each node also contains information about the uses of variables coming from transformations, definition of *key*/*values*, and the output.

USE nodes: It contains only the use of a variable *var* coming from a *key*/*value* transformation. A transformation occurs when a variable is formed by information coming from *key*, part of *key*, all/part of *values*, a unique *value* or combinations of the above. A sequence of these elements of *keys* and *values* is labeled in the node and represents a transformation between the input *key*/*values* variable and another variable.

Given a variable *var*, a statement *n* and a transformation *seq*, *P-USE-TRANS(var, n, seq)* is defined when variable *var* is used in the conditional statement *n* and comes from a transformation *seq*; and *C-USE-TRANS(var, n, seq)* when *var* is used in a non-conditional statement. The *seq* label contains the transformation of *var* in a sequence of *key*/*values* with conjunction $\wedge$ and disjunction $\vee$ connectors. The conjunction connector indicates that a transformation exists with both elements of the sequence, and the disjunction connector indicates that several transformations exist, one for each part of the sequence. For example, *P-USE-TRANS(var, 6, (key* $\wedge$ *value)* $\vee$ *key)* means that the variable *var* is used in the conditional statement 6 with two possible transformations, one is formed by the *key* and *value*, and the other only by the *key*. Because the transformation can be formed by parts of *key*/*values*, the *seq* sequence uses the following expressions:

- *Key* transformations:
  - [K]: Transformation over the whole *key*. For example: var = key, or var = key.length().
  - Ki: Transformation over the part *i* of *key*. Sometimes the *key* is composed of several elements. For example if the program should obtain the counting of every word in every year, the *key* is the compound of word and year. A transformation that involves the *key* part "word" (Kword) could be: var = getWord(key).

```
0  Reduce (Key key, List values){
1    sum = 0;
2    while (values.hasNext()){
3       sum += values.next();
4    }
5    emit(key, sum);
6  }
```



DEF-K(key, 0) DEF-V(values, 0)

P-USE-TRANS(values, 2, [V])

C-USE-TRANS(values, 3, [V])

EMIT({key}, {sum}, 5)

C-USE-TRANS(key, 5, [K])

C-USE-TRANS(sum, 5, [V])

Figure 1. MRFlow graph of WordCount program.

- *Values* transformations:
  - [V]: Transformation over several *values*. For example: var = values[0] + values[1].
  - V: Transformation over one *value*. For example: var = values.next().
- *Values* transformations with *categories*: The *Reduce* function could receive several *values* of a different nature and handle them in a different way. Such different *values* are considered a *category* and could come from different *Map* functions, a different data source or contain very different information. For example, a SPAM detector that receives several types of messages as a *values* (sms and email) has two categories: V:sms and V:email. The character of the sms and email, and the processing in the program is very different, so there are two categories.
  - [V:cat]: Transformation over several *values* of *cat category*. For example, the statement var = values[0] + values[1] could be a [V] transformation, but if values[0] and values[1] are from *category* sms, then the transformation is [V:sms].
  - V:cat: Transformation over one *value* of *cat category*. For example: if(isSms(values[0])) var = values[0].

DEF nodes: It contains the assignation of new content in the input *key* or in the list of *values*. Given a variable *var* and a statement *n*, *DEF-K(var, n)* is defined when new content is assigned to the variable *var* in the statement *n*, and *var* is the input *key* variable. *DEF-V(var, n)* is defined when *var* is the input *list(values)* variable.

Emit Nodes: The *Reduce* output is emitted by a special statement in <*key, value*> pairs. Given the variables $\{k_1,k_2,...k_m\}$, the variables $\{v_1,v_2,...,v_p\}$ and a statement *n*, *EMIT*($\{k_1,k_2,...k_m\}$, $\{v_1,v_2,...,v_p\}$, *n)* is defined when the *n* emits a <*key, value*> pair, the *key* is created by the variables $\{k_1,k_2,...k_m\}$, and the *value* by $\{v_1,v_2,...,v_p\}$.

As an example consider the *Reduce* function of *Wordcount* program [3] that counts the occurrences of each word. Figure 1 illustrates the *MRFlow* graph. The *Reduce* function receives a word as a key, and a list of numbers of occurrence as values, for instance <*hello, [1,1,1]*> means that the word "hello" has 3 occurrences in the text. In this program, the variables *key*, *values* and *sum* come from a transformation of *key*/*values* input variables. If the statement 3 is reached the *values* variable is transformed into *sum* by the addition of all *values* [V], but in other cases *values* is not transformed. The graph contains in node 0 the definition of *key* and *values*. The node 1 is empty because the *sum* variable is not created from *key*/*values* at this point. The node 2 contains a conditional statement of *values* variable. In node 3 there is a transformation of *values* in *sum*, and finally in node 5 the output, which contains *key* and *sum*, is emitted. The program does not combine *key* and *values* in any variable, and each *value* only represents the number of occurrences, so the program has neither categories nor connectors in the sequence of transformation (*seq* label).

3.2 Derivation of Test Cases

The goal of *MRFlow* is to derive tests in order to analyze the different *key*/*value* transformations with or without categories. In *MRFlow* graph, the paths under test start in definition of *key*/*value* and finish in each possible last transformation of such variables. Unlike *data flow* test criteria where each path is covered by a test case, in *MRFlow* for each path under test several situations to be covered (test coverage items) are defined and represent the transformations which are the goal of the test cases. Then the test cases are designed to cover the test coverage items in the path under test.

Transformation paths (*tp*): The paths under test, called transformation paths (*tp*), are extracted from transformations between input and output in *MRFlow* graph. One *tp* is created between each *DEF-K*/*DEF-V* node and *C-USE-TRANS*/*P-USE-TRANS* of each last transformation of *key* or *list(values)*. In the case of *DEF-K/DEF-V* to *P-USE-TRANS(var, n, seq)*, instead of creating one *tp*, several *tp* are created following all of the next nodes after the conditional statement *n*, as in other *data flow* test criteria [25]. For example, the transformations and *tp* of *WordCount* [3] program are represented in Figure 2. The program has 5 *tp* obtained from the transformation between *values* and *sum* (tp_1), the non-existence of *values* transformations (tp_2, tp_3 and tp_4) and the non-existence of *key* transformations (tp_5). The *values* variable is defined in node 0 and the last transformations are *sum* and *values* depending on whether statement 3 is reached or not. The sequence of transformation (*seq* label) between *values* and *sum* is [V] because it involves all values. In the case of *key* there is no transformation, so *key* is the last transformation. Finally, the transformation paths are obtained between *DEF-K/DEF-V* and *C-USE-TRANS/P-USE-TRANS* of last transformations. In the case of *P-USE-TRANS* like *P-USE-TRANS(values, 2, [V])*, one *tp* is created following the next nodes after node 2, that is node 3 (tp_2) and node 5 (tp_3).

Test coverage items: Each *tp* represents the transformations and the uses of transformation variables. Depending on the type of transformation (*key*, part of *key*, *values*, *value* or combination)

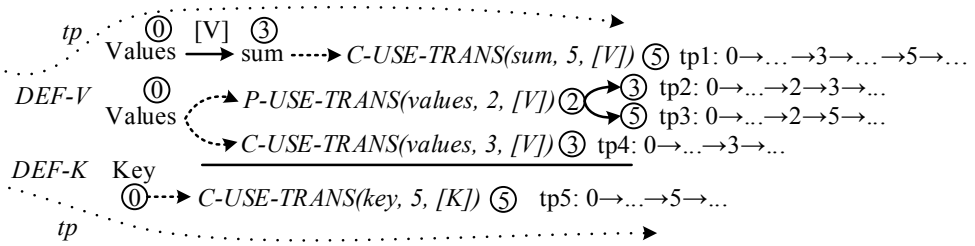


Figure 2. Example of transformation paths (tp) in WordCount program.

Table 1. Summary of program features and test results

	WordCount (Reduce)	**IPContry (Reduce)**
Number of transformations	3 (*Key*:1, *Values*:2)	3 (*Key*:1, *Values*:2)
DEF-K/DEF-V nodes	2 (*Key*:1, *Values*:1)	2 (*Key*:1, *Values*:1)
C-USE-TRANS nodes	3 (*Key*:1, *Values*:2)	5 (*Key*:2, *Values*:3)
P-USE-TRANS nodes	1 (*Key*:0, *Values*:1)	1 (*Key*:0, *Values*:1)
EMIT nodes	1	2
Transformation paths (*tp*)	5 (*Key*:1, *Values*:4)	6 (*Key*:2, *Values*:4)
Test coverage items	30 (*Key*:6, *Values*: 24)	30 (*Key*:6, *Values*:24)
Number of test cases	2	2
Test coverage items covered	16 (*Key*:4, *Values*:12)	16 (*Key*:4, *Values*:12)
Test coverage items not covered	14 (*Key*:2, *Values*:12)	14 (*Key*:2, *Values*:12)

several situations have to be tested. These situations (test coverage items) are usual in these types of programs and for each *tp* are defined next:

- Existence of information: *tp* created with empty data or non-empty data. Depending on the type of transformation (*seq* label in *MRFlow* graph) can occur:
  - If *tp* contains [V]: for each *category cat*, the transformation is created with *cat* data, or without *cat* data.
  - If *tp* contains [K]: the transformation is created with data in all *key*, or with empty data for each part of *key*.
- Validation: *tp* created with valid data or non-valid data. Depending on the type of transformation (*seq* label in *MRFlow* graph) can occur:
  - If *tp* contains [V]: for each *category cat*, the transformation is created with valid *cat* data, or non-valid *cat* data.
  - If *tp* contains [K]: the transformation is created with valid data in all *key*, or with non-valid data for each part of *key*.
- Output: *tp* reaches *EMIT* node or not.

Consider the *Reduce* function in the *WordCount* example (Figure 1). The test cases are designed in order to cover the test coverage items in each *tp*. For example, the test coverage items in all *tp*: "transformation with non-empty data", "with valid data" and "with output emission", can be covered by a test case with *Reduce* input *<hi, [1,1]>* which means that the word "hi" is repeated twice. In order to cover the other test coverage items (transformation with non-valid *key*, with empty values, and so on), new test cases have to be created, but it is possible that some test coverage items cannot be covered, as for example "Transformation without output emission" in all *tp* of *WordCount* because the *EMIT* node is always reached.

4. CASE STUDIES

In order to explore the applicability of the testing technique, *MRFlow* is applied over two popular programs: *WordCount* [3] which counts the occurrences of each word in a text, and *IPCountry* [24] which counts the number of IPs (Internet Protocol addresses) in each country. The goal of both programs is to count elements represented by the key. Further, in both programs the value is a list of numbers and the functionality consists of adding the elements of the lists. In *WordCount* the *key* is each word and the *value* represents the occurrence of the word, and in *IPCountry* the *key* is each country and the *value* represents the existence of IPs associated with the country.

For each program an *MRFlow* graph is created, from which the *tp* are extracted, then the test coverage items are derived, and finally the test case is created. The information of each step is summarized in Table 1, and in brackets is the information relative to the *key* transformations and *values* transformations. The first part focuses on the *MRFlow* graph, the second part summarizes the test coverage items, and in the third part the test case results

are described. In the *MRFlow* graph of both programs, the *<key, list(value)>* input variables has one definition and the program contains 3 transformations: transformation of *values* into another variable, no *values* transformation and no *key* transformation. Then the *C-USE-TRANS*/*P-USE-TRANS* are created from these variables: 1 *P-USE-TRANS* in each program, 3 *C-USE-TRANS* in *WordCount* and 5 in *IPCountry*. In the graph, finally, the *EMIT* nodes are created from each emission statement.

From the above graph, the transformation paths (*tp*) are obtained, and then for each *tp* the test coverage items are derived. The *Wordcount* has 5 *tp* and *IPCountry* has 6 *tp*, but in both cases there are 30 test coverage items.

It is not possible to cover 14 of the test coverage items due to some program constraints such as it is impossible to create *values* with empty content, the node *EMIT* is always reached, and so on. The rest of the test coverage items, 16, are covered with two test cases: *<hi, [1,1]>* and *<hello,, [1,1]>* (hello with a comma) for *WordCount*, and *<Spain, [1,1,1]>* and *<###, [1,1,1]>* for *IPCountry*.

The test cases detect two defects because of the non-validation of *key*. If *WordCount* program receives "hello, hello, hello", the expected output is hello:3, but the real output is hello:1, hello,:2 because the *Reduce* function receives an invalid *key* "hello," that is not a word. In *IPCountry* the program fails when it receives a non-country as *key*, for example *Reduce* receives *<###, [1,1,1]>* in the test case and the expected output is nothing because "###" can be an unexpected log/exceptional data but it is not a country.

The two defects found in the programs are caused by the non-validation of input data together with exceptional/non-valid data. In these two programs, *MRFlow* allows to test the functionality with a few test cases that cover many test coverage items.

5. CONCLUSIONS

The *MapReduce* development and programs contain characteristic defects such as the incorrect validation or incorrect processing of different types of data. These defects produce a failure when the *key* or the *values* contain some data that is not correctly processed in the *MapReduce* programs. In this work, the testing technique *MRFlow* is introduced in order to test the *MapReduce* programs. *MRFlow* is based on data flow test criteria and analyzes the program transformations under several situations to cover. This testing technique is applied over two popular programs and with two test cases covers several situations in the transformations which reveal one defect in each program. The faults are caused by the non-validation of *key*, but *MRFlow* in other programs could detect other defects relative to the transformations of *keys* and *values*.

As future work we plan to apply *MRFlow* in more programs and to automate the technique in areas such as test coverage items, the execution of test cases, the derivation of test cases or the graph on which these test cases are derived.

6. ACKNOWLEDGMENTS

This work was supported in part by project TIN2013-46928-C3-1-R, funded by the Spanish Ministry of Science and Technology, and GRUPIN14-007, funded by the Principality of Asturias (Spain) and ERDF funds.

7. REFERENCES

[1] Hadoop: open-source software for reliable, scalable, distributed computing. http://hadoop.apache.org/ Accessed May, 2015.

[2] Institutions that are using hadoop for educational or production uses. http://wiki.apache.org/hadoop/PoweredBy Accessed May, 2015.

[3] Wordcount 1.0. http://hadoop.apache.org/docs/r2.7.0/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example:_WordCount_v1.0 Accessed May, 2015.

[4] IEEE draft international standard for software and systems engineering–software testing–part 4: Test techniques, 2014.

[5] Alshahwan, N., and Harman, M. State aware test case regeneration for improving web application test suite coverage and fault detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (2012), ACM, pp. 45–55.

[6] Bertolino, A. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering* (2007), IEEE Computer Society, pp. 85–103.

[7] Camargo, L. C., and Vergilio, S. R. Classicação de defeitos para programas mapreduce: resultados de um estudo empírico. In *AST - 7th Brazilian Workshop on Systematic and Automated Software Testing* (2013).

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

[9] Chen, Y., Ganapathi, A., Griffith, R., and Katz, R. The case for evaluating mapreduce performance using workload suites. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on* (2011), IEEE, pp. 390–399.

[10] Csallner, C., Fegaras, L., and Li, C. 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* (2011), ACM, pp. 504–507.

[11] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[12] Dörre, J., Apel, S., and Lengauer, C. Static type checking of hadoop mapreduce programs. In *Proceedings of the second international workshop on MapReduce and its applications* (2011), ACM, pp. 17–24.

[13] Gudipati, M., Rao, S., Mohan, N. D., and Gajja, N. K. Big data: Testing approach to overcome quality challenges. *Big Data: Challenges and Opportunities* (2013), 65–72.

[14] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on* (2010), IEEE, pp. 41–51.

[15] Kavulya, S., Tan, J., Gandhi, R., and Narasimhan, P. An analysis of traces from a production mapreduce cluster. In

Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on (2010), IEEE, pp. 94–103.

[16] Kim, K., Jeon, K., Han, H., Kim, S.-g., Jung, H., and Yeom, H. Y. Mrbench: A benchmark for mapreduce framework. In *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on* (2008), IEEE, pp. 11–18.

[17] Kocakulak, H., and Temizel, T. T. A hadoop solution for ballistic image analysis and recognition. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on* (2011), IEEE, pp. 836–842.

[18] Li, N., Escalona, A., Guo, Y., and Offutt, J. A scalable big data test framework. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on* (2015), IEEE, pp. 1–2.

[19] Li, S., Zhou, H., Lin, H., Xiao, T., Lin, H., Lin, W., and Xie, T. A characteristic study on failures of production distributed data-parallel programs. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 963–972.

[20] Mattos, A. J. d. Test data generation for testing mapreduce systems. Master's thesis, Universidade Federal do Paraná, 2011.

[21] Mittal, A. Trustworthiness of big data. *International Journal of Computer Applications 80*, 9 (2013), 35–40.

[22] Morán, J., De La Riva, C., and Tuya, J. Mrtree: Functional testing based on mapreduce's execution behaviour. In *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on* (2014), IEEE, pp. 379–384.

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

[24] Owens, J. R., Femiano, B., and Lentz, J. *Hadoop Real World Solutions Cookbook*. Packt Publishing Ltd, 2013.

[25] Rapps, S., and Weyuker, E. J. Selecting software test data using data flow information. *Software Engineering, IEEE Transactions on*, 4 (1985), 367–375.

[26] Ren, K., Kwon, Y., Balazinska, M., and Howe, B. Hadoop's adolescence: an analysis of hadoop usage in scientific workloads. *Proceedings of the VLDB Endowment 6*, 10 (2013), 853–864.

[27] Schatz, M. C. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics 25*, 11 (2009), 1363–1369.

[28] Sharma, M., Hasteer, N., Tuli, A., and Bansal, A. Investigating the inclinations of research and practices in hadoop: A systematic review. In *Confluence The Next Generation Information Technology Summit (Confluence), 2014 5th International Conference-* (2014), IEEE, pp. 227–231.

[29] Sneed, H. M., and Erdoes, K. Testing big data (assuring the quality of large databases). In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on* (2015), IEEE, pp. 1–6.