This paper is a post-print paper accepted in “IEEE International Conference on Software Quality, Reliability and Security (QRS), 2017“

The final version of this paper is available through IEEE Xplore in the next link: http://ieeexplore.ieee.org/document/8009910/

J. Morán, A. Bertolino, C. de la Riva and J. Tuya, "Towards Ex Vivo Testing of MapReduce Applications," 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, 2017, pp. 73-80. doi: 10.1109/QRS.2017.17

Towards Ex Vivo testing of MapReduce applications

Jesús Morán
Department of Computing
University of Oviedo
Gijón, Spain
moranjesus@uniovi.es

Antonia Bertolino
ISTI-CNR
Consiglio Nazionale delle Ricerche
Pisa, Italy
antonia.bertolino@isti.cnr.it

Claudio de la Riva, Javier Tuya
Department of Computing
University of Oviedo
Gijón, Spain
{claudio, tuya}@uniovi.es

***Abstract*—Big Data programs are those that process large data exceeding the capabilities of traditional technologies. Among newly proposed processing models, MapReduce stands out as it allows the analysis of schema-less data in large distributed environments with frequent infrastructure failures. Functional faults in MapReduce are hard to detect in a testing/preproduction environment due to its distributed characteristics. We propose an automatic test framework implementing a novel testing approach called Ex Vivo. The framework employs data from production but executes the tests in a laboratory to avoid side-effects on the application. Faults are detected automatically without human intervention by checking if the same data would generate different outputs with different infrastructure configurations. The framework (MrExist) is validated with a real-world program. MrExist can identify a fault in a few seconds, then the program can be stopped, not only avoiding an incorrect output, but also saving money, time and energy of production resources.**

Keywords—software testing; automatic testing; ex vivo testing; metamorphic testing; Big Data; Hadoop; MapReduce

I. INTRODUCTION

The *Big Data* field involves the recent trends in data analysis that go beyond the capabilities of the traditional technology [1]. Although these programs are considered critical for several Fortune 1000 enterprises [2], there are several challenges and concerns: poor data quality [3], [4], lack of technological skills [5], [6], and different technological issues such as, among others, complexity [7], maturity [8], operability [9] and technical problems [3]. Some of the previously stated problems complicate the construction of the projects and could lead to failures. Gartner forecasts that in 2017 60% of the *Big Data* projects fail to go beyond piloting and will be abandoned [10].

The *MapReduce* processing model [11] stands out among *Big Data* projects, as it allows the analysis of large datasets in a distributed architecture. *MapReduce* programs are broadly used [12], and are supported by several frameworks such as Hadoop [13], Flink [14] and Spark [15], among others. These frameworks manage the execution of the *MapReduce* applications allocating resources and dealing with the frequent infrastructure failures [16].

A study by Kavulya et al. [17] reveals that around 3% of *MapReduce* programs in production do not finish their execution, whereas another broader study places the percentage between 1.38% and 33.11% [18]. The quality of the *MapReduce* programs is important, especially those employed in critical sectors such as health (DNA alignment [19]) and security (image processing in ballistics [20]).

In a previous work, we identified and classified several *MapReduce* faults [21], and we proposed a testing technique to detect them in an automated way just by varying the test input data [22]. This paper improves on these works with an automatic testing framework to detect the faults when the programs are deployed and executed in production. As we discuss in Section IV, to detect faults that may depend on the deployed *MapReduce* configuration, one would need to test the application in production (In Vivo), but this is hardly feasible due to a lack of control on the tester's side. We propose here a hybrid approach between testing in the laboratory and testing in production, which we have named the *Ex Vivo* approach: the tests are automatically obtained from the runtime data, but they are executed outside of the production environment so as not to affect the application. The proposed framework can automatically detect functional faults without requiring any human intervention.

In summary, the main contributions of this paper are:

1. A novel generally-applicable test approach called Ex Vivo, and its adaptation to *MapReduce* programs. The test cases are designed based on production information such as runtime data, but the tests are executed outside the production environment to obtain fine-grained control and not to impact negatively on the execution of applications.
2. The automation of the previous approach in a continuous testing framework, called MrExist. When a user executes a program in production, the framework automatically performs testing with runtime data outside the production environment. If the framework detects functional faults, the user is notified so that the program can be stopped not only to avoid incorrect output, but also to save time, energy and cost of the *Big Data* resources employed in potentially faulty executions.
3. The validation of the previous approach within a real-world case study.

The remainder of the paper is organized as follows. Section II overviews the *MapReduce* processing model. Section III discusses related work. Section IV defines the Ex Vivo testing

approach, and Section V describes the MrExist framework based on this approach that is adapted to the *MapReduce* processing model. Then MrExist is validated in Section VI. Finally, the conclusions and plans for future work are included in Section VII.

II. MapReduce

The *MapReduce* processing model follows the divide and conquer principle with two functionalities: Mapper that divides one problem in several subproblems, and Reducer that solves each subproblem. Both functionalities receive and generate the data in a <key, value> pair fashion. The key represents the subproblem, and the value contains the data to solve this subproblem.

Consider as an example a program that calculates the average temperature per year. This problem can be divided in as many subproblems as there are years, then each subproblem only calculates the average temperature for one year. To start off, several Mappers receive subsets of historical data and emit <year, temperature of this year>. After the execution of all Mappers, the temperatures (values) are grouped by their year (key). Then, several Reducers receive subproblems like <year, [all temperatures of this year]>, that is one year with all temperatures for this year, and emit the average.

For example, Fig. 1 represents the execution of the following temperature data: year 2000 with 3º, 2002 with 4º, 2000 with 1º, and 2001 with 5º. The first two inputs are analyzed in one Mapper task and the remainder in another task. After the execution of the Mappers, 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 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 executes the program, it should generate the expected output.

The *MapReduce* programs process high quantities of data between Mapper and Reducer. In order to optimize the program, a Combiner functionality can be implemented. This functionality is run after the Mapper with the aim of removing the irrelevant <key, value> pairs to solve the subproblem. In *MapReduce,* other components can also be implemented, such as for example Partitioner that determines for each <key, value> pair which Reducer analyses it, Sort that controls the order of <key, value> pairs, and Group that aggregates the values of each key before the Reducer.

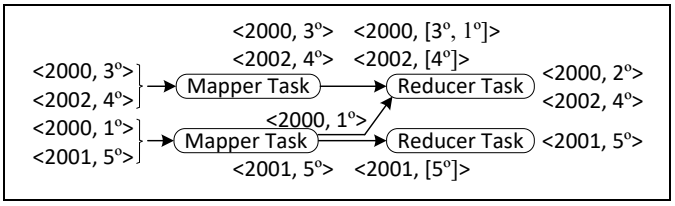

Fig. 1. Execution of MapReduce program that calculates the average temperature per year

An incorrect implementation of these functionalities could cause a failure in one of the different ways in which Hadoop or other distributed systems can execute the program. These functional faults are difficult to detect during testing because they are triggered in aggressive environments with a mix of large data and computer failures, among others.

III. Related Work

Software testing is one of the quality assurance techniques most used to evaluate software products [23]. This field has experimented great progress in recent years [24], but there are still some challenges to test *Big Data* programs [25], [26]. Despite the fact that most works are focused on performance testing [27], [28], functional testing is also important [29].

Some functional faults depend on how the distributed system, for example Hadoop, executes the programs according to the infrastructure configuration. If the program generates incorrect output in some configurations and the expected output in others, then the program has a functional fault. A study of 507 *MapReduce* programs in production reveals at least 5 different kinds of these faults [30]. To detect them, Csallner et al. [31] and Chen et al. [32] use testing techniques based on symbolic execution and model checking. Other authors [21], [33] identified and classified more kinds of these faults that are caused by the infrastructure configurations. To detect them, Morán et al. [34] designed an automatic testing technique based on combinatorics and simulation. This paper also aims to detect these kinds of faults that are caused by the infrastructure configurations, but not executing the tests in stages prior to production. Instead, the approach for testing proposed in this paper is automatically performed during production and taking advantage of the runtime data.

In the production environment, the infrastructure failures are fairly frequent. Several research lines suggest the injection of infrastructure failures [35], [36] during the testing, and several tools support it [37]–[39]. For example, Marynowski et al. [40] create test cases specifying which computers fail and when. Some faults can be detected with the injection of infrastructure failures, but others require a full control of the distributed system and its underlying large infrastructure. To detect this kind of faults, this work does not inject infrastructure failures, but simulates the different infrastructure configurations in a lab to obtain fine-grained control and reproducibility of the tests.

Other testing techniques focus on the generation of test input data with different approaches: using data flow [41], based on a bacteriological algorithm [42], or with input domain together with combinatorial testing [43]. Unlike the previous testing techniques, this paper takes the data directly from production in runtime.

There are several tools to design and execute test cases for *MapReduce* applications. Herriot [44] allows the execution of the tests in a real cluster at the same time that it supports the injection of the infrastructure failures. Other tools called MiniClusters [45] execute the test cases in a cluster simulated in memory. For unit testing, MRUnit [46] provides an adaptation of JUnit [47] to the *MapReduce* processing model. This paper also proposes a testing tool for the *MapReduce*

processing model, but unlike the others it performs full automatic testing in production with the runtime data.

IV. Ex Vivo Testing

Modern software applications are increasingly distributed, pervasive, and adaptive. For such systems, the boundary between development-time and production is vanishing [48], and several authors (e.g., [49], [50]) have proposed that software testing can (or should) be used even after deployment to continue detecting functional faults that cannot be found in the development environment. Testing in the production environment has been referred to with different terminology [51]:

- Online testing, as opposed to offline, to highlight that testing is done without interrupting the normal operation.
- Testing "in the field", as opposed to traditional testing performed "in the lab".
- Runtime testing, to highlight that testing is done employing execution data from operation, rather than other artificial data.
- A form of testing in production is also monitoring, which is referred to as passive testing in contrast with actively providing some stimulus (test input).

From the previous approaches, other testing approaches arise, among them In Vivo testing [52]. This kind of testing is performed inside the production environment but in an isolated process in order not to affect the program executed in production. In this way, testing can take advantage of information from production such as runtime data, third party libraries or configurations. However, online tests also consume memory and other production resources that could negatively impact the program executed in production, especially regarding performance.

The performance of the *MapReduce* programs is important because they usually analyze large and complex datasets [53]. The information of these datasets can be useful for carrying out testing in runtime [54], but the execution of the tests in the production environment is problematic for several reasons. Hadoop automatically manages the executions carried out in production, but does not support fine-grained control and reproduction in the same circumstances. In addition, the tests executed in production consume resources and can negatively impact the performance of the applications. Although production data can be a good test data, in the *MapReduce* context it is not feasible to execute the test cases in the production environment like the In Vivo testing. A more convenient alternative is the execution of the test cases in a simulator outside production, but using production data as test inputs.

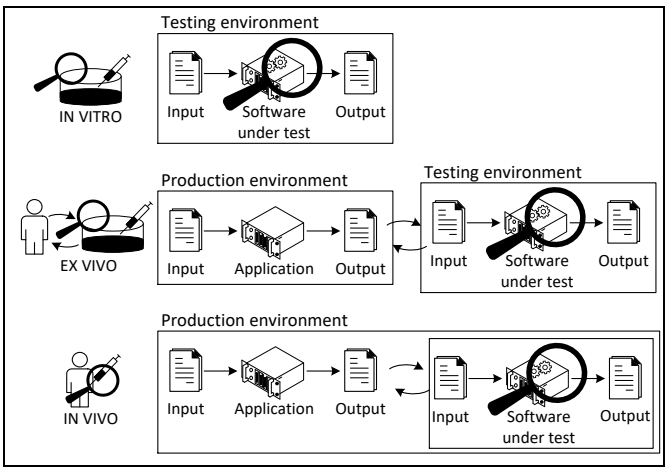

Fig. 2. Software testing approaches

Thus, this paper proposes a novel testing approach called Ex Vivo. This new type of testing takes some information from production like the In Vivo approach, but performs testing outside the production environment. Then testing can take advantage of the runtime information but in a more controllable way than In Vivo, and without the limitations imposed by the actual production environment. The Ex Vivo testing also has few risks to impact the execution of programs because it does not take resources from production like the In Vivo approach. To the best of our knowledge there are no testing approaches with these principles, either in *MapReduce* or in other software contexts.

The terminology used of In Vivo and Ex Vivo testing, together with In Vitro, has been borrowed from biological sciences where they are used to denote different kinds of tests. As Fig. 2 describes, In Vivo (latin for "within the living") are those tests performed inside an organism, Ex Vivo (latin for "out of living") outside, and In Vitro in a tube. In software testing the organism could be seen as the analogy for the production environment whereas the tube could be the development-testing environments. Consequently, In Vitro is the traditional testing that does not take advantage of the production information to detect faults. In contrast, both In Vivo and Ex Vivo, take advantage of this information (for example runtime data), but testing is performed in different environments: In Vivo performs testing inside the production environment, while Ex Vivo performs testing outside.

V. MrExist: Ex Vivo Testing Framework For MapReduce Applications

In order to perform testing not only in development-testing phases, but also actively during production, an automatic continuous testing framework is proposed. This framework is called MrExist (MapReduce EX vIvo teSTing) and it is based on the Ex Vivo testing approach exposed in Section IV adapted to the *MapReduce* processing model characteristics. These programs are executed in a non-deterministic way by a distributed system, for example by Hadoop. Usually these systems do not allow fine-grained control of the execution, thus making testing more difficult. The Ex Vivo framework proposed here takes data under execution and then executes test cases based on such real data in a test server outside the production environment. Therefore the execution of the test cases does not take resources from production, does not introduce side-effects and can be fully controlled.

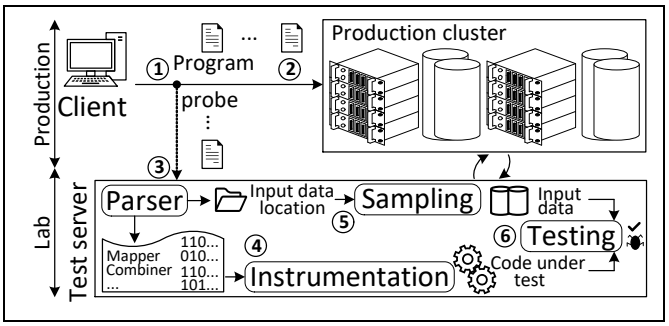

Fig. 3. Architecture of MrExist (MapReduce EX vivo teSTing)

Fig. 3 describes the MrExist framework starting with a user that executes a program and finishing by testing performed automatically without any knowledge either of the specification or of the expected output. Firstly a probe detects that a user has executed a program (1), and the probe sends this program to the test server (2). Then the program is parsed by the test server (3) to obtain the location of the data and the *MapReduce* functionality code. This code is instrumented (4) to analyze its internal states, and the test input data are sampled from the production data (5). Finally, the testing is performed using a specific *MapReduce* testing technique [22], [34] that only needs the test input data and the program to detect functional faults (6).

Generally speaking, a challenge for testing in production is how to check whether the output is correct or not (test oracle), especially in those programs that are specifically developed to obtain some previously unknown or costly answer [55], as for example some machine learning programs. Such problems do not exist in our approach, as we can compare the outputs obtained for the same data in different configurations. The MrExist framework automatically detects a functional fault when the same data executed in different configurations do not generate an equivalent output. The different parts of MrExist are described in detail in the subsections below together with the following example.

Consider the program that calculates the average temperature per year described in Section II and extended with a Combiner to improve the performance. The Combiner receives several temperatures and then they are replaced by their average to decrease the data sent from one computer to another. This program has a functional fault because all the temperatures are needed to obtain the total average temperature. First, the Combiner replaces the data available locally for their average, and then the Reducer calculates the global average with these local averages, but sometimes this output does not match the average of all temperatures. This kind of fault is difficult to detect in the *MapReduce* programs and is usually masked during the testing [22] because the latter does not suffer aggressive situations as in the execution of large data in production like parallelization, computer failures, automatic optimizations and so on. Then these programs can be released to production and the Ex Vivo framework proposed, MrExist, could automatically detect faults and notify the user in runtime.

A. Parser

The probe sends the program executed in production to the test server. Then the program is parsed in order to obtain the *MapReduce* code functionality and the location of the dataset employed in production. The parser analyzes the bytecode with Javassist [56] not only to obtain the bytecode of the Mapper, Combiner and Reducer, but also other *MapReduce* advanced functionalities such as Partitioner and Sort, among others that are relevant for testing.

The parser employs a cache based on MD5 hashes [57] that leverages the communications between client and test server. The client only sends a few bytes of hash instead of the program, and when the test server does not have the program in the cache, then it can request it. The parser also detects automatically if the program under test has been tested before or not, and then registers the different versions/improvements of the program based also on its hashes.

For example, when the user executes the program that calculates the average per year, the probe sends the MD5 hash of the program to the test server. If the program is not in the cache, the test server requests the program from the probe. Then the program is parsed in the test server obtaining (1) location of dataset, (2) code of the *MapReduce* functionality, and (3) other metadata such as the number of the version. For the program under test the parser obtains the following *MapReduce* code: AvgMapper function (Mapper), AvgReducer function (Reducer and Combiner), TextInputFormat function (Input format), among other advanced codes of the *MapReduce* programs and dependencies. Finally, the parser checks if the program has been tested before or if it is a new version with changes of a previous program. Then the parser registers this information about the program version, allowing the visualization of the quality evolution in the user programs.

B. Instrumentation

The Mapper, Combiner and Reducer functions in Hadoop do not return any data, the <key, value> pairs are sent from one function to another based on buffers, dumps, and remote calls, among others. In order to observe the internal states of the program under test, the *MapReduce* functions are instrumented. The instrumentation automatically adds mocks, stubs and spies inside the code using mocking frameworks widely used in practice [58] such as Mockito [59] and PowerMock [60].

For example, in the program that calculates the average temperature per year, the parser obtains that avgMapper and avgReducer code implement the Mapper, Combiner and Reducer. In order to enable full control and monitoring of their internal states during testing, these functions are instrumented with mocks, stubs and spies.

C. Sampling

In addition to the code under test, MrExist needs data to perform testing. The sampling method generates the test input data from the location previously obtained by the parser.

In *Big Data*, the datasets usually contain a huge amount of data stored in a distributed database or filesystem, such as HBase [61] or HDFS (Hadoop Distributed File System) [62].

In terms of resources, it is not feasible to perform functional testing with all of these large data. Instead, MrExist generates a smaller test input data with a reservoir sampling [63]. This algorithm samples streams of data and can be parallelized to improve the performance. The MrExist framework implements the sampling using the *MapReduce* processing model to employ *Big Data* power during the sampling of the large datasets. This algorithm assigns a random number to each <key, value> pair, and then only the highest are sampled.

The samples obtained from the sampling algorithm are used as test input data and are saved in a specific binary format for the <key, value> data, called SequenceFile [64]. These samples are obtained based on randomness, but the algorithm also supports pseudorandom numbers, also called seeds, to obtain the samples in a deterministic way and support the reproduction of the test cases in the same circumstances.

In a *Big Data* cluster there are several datasets, but the majority of the programs only analyze the same one, two or few datasets [18], and sometimes concurrently [65]. To avoid multiple samplings of these *Big Data* datasets a cache is implemented to improve performance [66], [67]. Then the sampling method is only executed when the dataset has no samples in cache. These samples can also be generated proactively, for example scheduling the samplings of the available datasets during weekends, nights or at other times with low production activities.

In the program that calculates the average temperature per year, the parser obtains the dataset used in production. Then MrExist checks if the cache contains test input data for this dataset. If there is no data, a sampling is performed obtaining, among others, the temperatures 4^{o}, 2^{o} and 3^{o} in the year 1999. Then these test input data are available in the cache for future uses in testing.

D. Testing

The execution of the program in production is managed by a distributed system, for example Hadoop, that automatically allocates resources in a parallel way, re-executes different parts of the program in case of computer failures, performs some data optimization and mixes the analysis of different parallel traces, among others. These automatic mechanisms guide the execution in a highly scalable way, but could also cause that a program generates an incorrect output. In this case, the program has a functional fault because it generates valid or incorrect output depending on the infrastructure configuration [21].

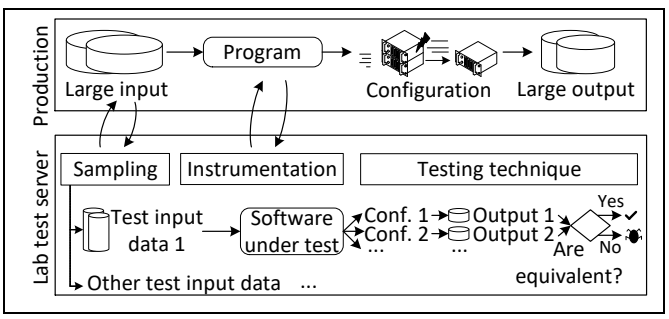

Fig. 4. Testing technique used in MrExist

MrExist detects these faults employing a specific *MapReduce* testing technique [22]. This testing technique executes the same data in different infrastructure configurations and checks whether their outputs are similar or not. These infrastructure configurations are generated with a combination of a different number of Mapper/Reducer tasks, and several *MapReduce* optimizations, among others. Fig. 4 describes the execution of the testing technique taking advantage of the sampling and instrumentation of the previous sections. The test server obtains the test input data from sampling, and the software under test from instrumentation. Then the test server executes each test input data with different configurations and finally checks if the outputs are equivalent, revealing a fault if they are not. These configurations are generated and executed with an extension of MRUnit [22], [46] (JUnit [47] for *MapReduce*), and checked with Hamcrest matchers [68].

In the program that calculates the average temperature per year, MrExist automatically detects a fault. First, the parser obtains that the program under test has a customized Mapper, Combiner and Reducer functionalities, among other *MapReduce* advanced functions. Then these functions are instrumented, and the testing is performed with the different test input data obtained from production by the sampling method. Fig. 5 describes the testing performed with the test input data: 4^{o}, 2^{o} and 3^{o} in the year 1999. The test server iteratively generates and simulates different configurations, and then checks if one of the outputs is not equivalent to the others. The first configuration generated is made up of only one Mapper, one Combiner and one Reducer, producing 3^{o} as output. The second configuration also generates 3^{o} with a different configuration: two Mappers (the first with two data, and the second with one data), two Combiners and one Reducer. But the third configuration generates a different output, 3.25^{o}, that automatically reveals a fault because the program generates different output depending on how it is executed. This configuration is composed by two Mappers (the first with one data, the second with two data), two Combiners and one Reducer. In this case, the program does not support this Combiner because it replaces the temperatures available locally by their average, and then Reducer calculates erroneously the total average with these local averages.

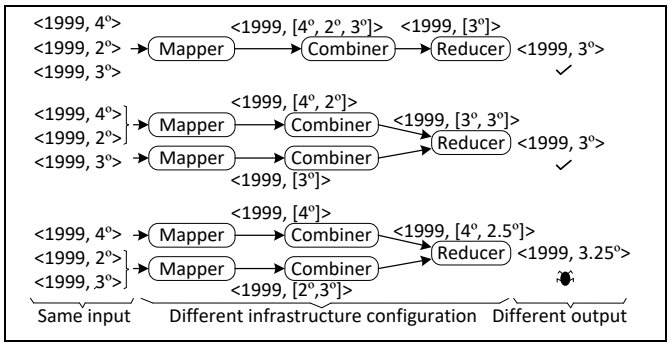

Fig. 5. Execution of MapReduce program in different infrastructure configurations

Once the fault is automatically detected, MrExist sends an email to the user in order to notify the fault. The email not only contains the existence of the fault, but also represents how this fault is caused, as can be seen at the bottom of Fig. 5. Then the user can stop the program to avoid incorrect worthless output while also saving money, energy and time of large-scale computation resources, especially for those *MapReduce* programs that finish their execution after several hours [18] or days [17].

E. Test oracle

The test oracles have some properties to characterize the testing efficacy [69], [70]. The MrExist framework aims to detect faults without human intervention, and the oracle used during testing is an automated partial oracle [55]. This kind of oracle can detect some faults without any knowledge about the expected output. The oracle employed in MrExist is automatically derived from the program executions [71] using metamorphic testing [72]–[74], that is a field also employed to test machine learning programs [75] and in In Vivo frameworks [76]. The metamorphic testing given a test case checks relationships inside one or different executions of the program. The test case is called original test case, the different executions are called follow-up test cases, and the relationship that should be satisfied is called metamorphic relationship.

The MrExist framework proposes a metamorphic testing that can automatically test the *MapReduce* programs. This approach obtains the test cases from production (original test cases) then executes them with different configurations (follow-up test cases) and finally checks if their outputs are equivalent (metamorphic relationship), if not a potential fault is detected.

In most metamorphic testing research, the test cases are generated with random testing [77]. In MrExist, the original test cases are also obtained randomly based on a sampling of the production dataset. One benefit of testing with this automatic oracle is that these random data can be useful to cover more test domains [78].

According to the study of Segura et al. [77] the number of metamorphic papers will increase in the following years, but to date 49% employ the metamorphic testing capabilities to different problem domains, and only 2% present a tool. In our case, this paper not only defines and automatizes the metamorphic relationship to the *MapReduce* domain, but also develops a tool that detects faults in production without human intervention and non-intrusively from runtime data.

F. Probe

MrExist executes testing with runtime data when a *MapReduce* program is executed in production. The probe detects the execution of the program and catches it together with other information about the context and user. Then the probe sends the program and all information to the test server asynchronously with the aim of minimizing the impact of the probe in terms of execution time.

The probe is not intrusive in the sense that no modification or additional code is necessary either in the *MapReduce* applications or in the production cluster. To enable MrExist framework it is only necessary (1) the replacement of one library in the Hadoop client that adds the probe for all programs executed in this computer, and (2) the deployment of the test server to perform testing with access to the Hadoop cluster and data sources employed in production. The test server is a Java application that automatically deploys a Jetty server [79] and serverless database SQLite [80], [81] both embedded inside. Thus the test server is self-contained and can easily be deployed from one computer to another in case of computer failures.

VI. Case Study

In order to validate the testing framework MrExist, we use the real-world program Open Ankus [82] as case study. This program implements Machine Learning and Data Mining libraries using the *MapReduce* processing model. One part of the program is a recommendation system that predicts the best books for each user based on the books read by others. The system obtains the similarities between users based on the points that each user assigns to different books. Given these similarities, the system predicts the points from each user to each book, and the highest are recommended. Finally, when the user assigns points to the book, the system calculates the error of its previous prediction.

This program is executed in the production environment, and MrExist automatically notifies the existence of a functional fault. This fault arises in the following situation: (1) the system predicts that Alice could assign 0 points to Don Quixote, (2) Alice assigns 0 points to Don Quixote, (3) later the system detects a change in Alice's taste and predicts that Alice could assign 10 points to Don Quixote, and (4) Alice assigns 10 points to Don Quixote. For the previous situation obtained from runtime data, the expected output is that the predictions are accurate with 0% of error. But MrExist detects that the *MapReduce* program has a fault because it sometimes obtains 100% of error as output and 0% in others, depending on the infrastructure configuration (number of computers, computer failures, and so on). The program checks per each user-book the first points assigned against the first points predicted, and so on (0 vs 0 and 10 vs 10, 0% of error). The fault arises when the infrastructure configuration causes that the input data are processed in a different order. The *MapReduce* processing model splits the input data into several subsets that are analyzed in parallel, then the final part of the input data can be processed before the first part. This fault is revealed when the infrastructure configuration causes that the first assignment is checked against the second prediction, and the second assignment against the first prediction (0 vs 10 and 10 vs 0, 100% of error).

Fig. 6 depicts the Ex Vivo testing for the previous situation. When the program is executed in production, the tests are executed in the test server. Firstly the large runtime data is sampled to obtain test input data, among others: (1) prediction of Alice-Don Quixote: 0 points, (2) assignment of Alice-Don Quixote: 0 points, (3) prediction of Alice-Don Quixote: 10 points, and (4) assignment of Alice-Don Quixote: 10 points. Then these runtime data are executed in several configurations. The first configuration obtains 0% of error as

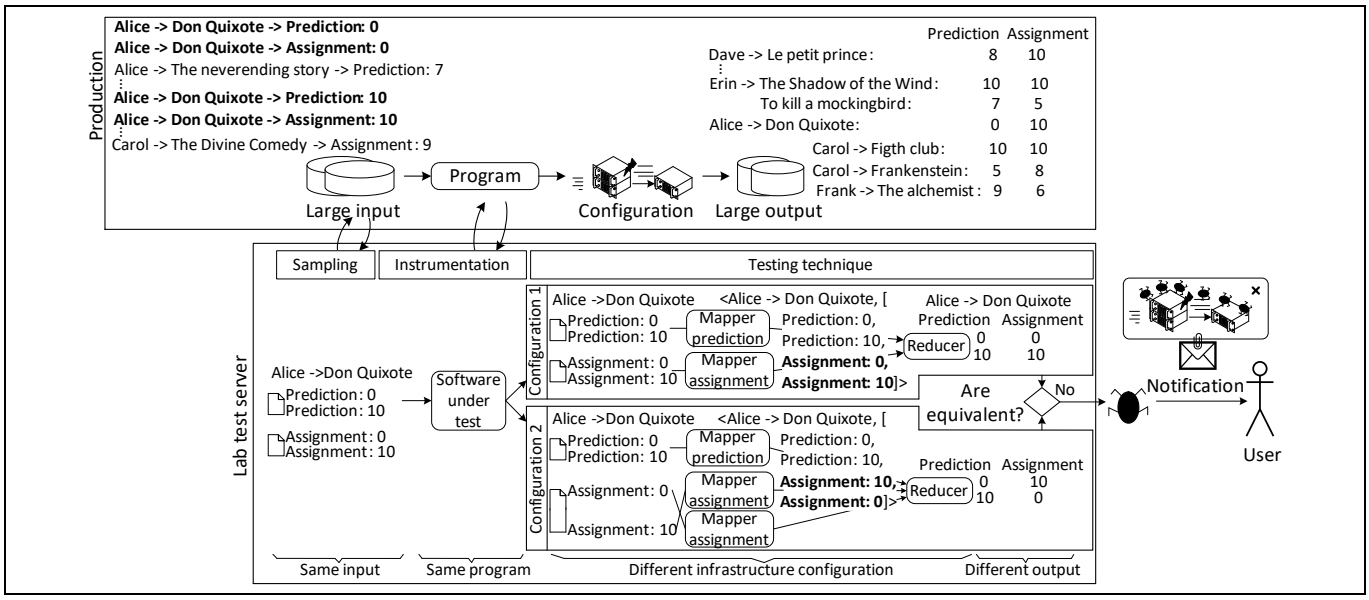

Fig. 6. Fault detected automatically by MrExist in the recommendation system of Open Ankus

output whereas the second obtains an incorrect output of 100% of error because the infrastructure configuration causes that the program analyzes the input data in a different order. Then the testing framework MrExist notifies the user of the existence of a functional fault in the program executed in production.

VII. Conclusions and Future Work

This paper introduces a context-independent testing approach called Ex Vivo to detect faults. The tests are designed from production data and executed in a different environment to avoid side-effects and gain fine-grained control. This approach is employed in an automatic testing framework for *MapReduce* programs. The execution of an application triggers the testing in background taking advantage of runtime data and detecting faults without human intervention. In the case of a fault, the framework notifies the user who can stop the faulty program, allowing to improve the quality, avoid incorrect output and save time, money and energy of the large-scale resources executed in production.

This approach is applied in a real-world program executed in a production cluster, and without any modification, the testing framework automatically notifies that the program has a functional fault.

As future work, we are creating an automatic method to forecast in runtime the percentage of the production output affected by the functional fault. Another research line pursues self-healing through automatic localization and removal of the fault in production.

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 POLOLAS (TIN2016-76956-C3-2-R), projects funded by the Spanish Ministry of Economy and Competitiveness; GRUPIN14-007, funded by the Principality of Asturias (Spain); GAUSS (PRIN 2015, 2015KWREMX), funded by Italian MIUR; and ERDF funds.

References

[1] ISO/IEC JTC 1 - Big Data, preliminary report. 2014.

[2] NewVantage Partners LLC, “Big Data Executive Survey 2016 An Update on the Adoption of Big Data in the Fortune 1000,” 2016.

[3] Xerox, “Big Data in Western Europe Today,” 2015.

[4] L. Cai and Y. Zhu, “The Challenges of Data Quality and Data Quality Assessment in the Big Data Era,” *Data Sci. J.*, vol. 14, p. 2, 2015.

[5] B. Marr, “Where Big Data Projects Fail,” 2015. [Online]. Available: http://www.forbes.com/sites/bernardmarr/2015/03/17/where-big-data-projects-fail/.

[6] Pure Storage, “BIG DATA’S BIG FAILURE: The struggles businesses face in accessing the information they need,” 2015.

[7] Capgemini Consulting, “Big Data survey,” 2014.

[8] V. Marx, “Biology: The big challenges of big data,” *Nature*, vol. 498, no. 7453, pp. 255–260, Jun. 2013.

[9] D. Bachlechner and T. Leimbach, “Big data challenges: Impact, potential responses and research needs,” in *2016 IEEE International Conference on Emerging Technologies and Innovative Business Practices for the Transformation of Societies (EmergiTech)*, 2016, pp. 257–264.

[10] Gartner, “How to Take a First Step to Advanced Analytics,” 2015.

[11] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Proc. OSDI - Symp. Oper. Syst. Des. Implement.*, pp. 137–149, 2004.

[12] Institutions that are using Apache Hadoop. [Online]. Available: https://wiki.apache.org/hadoop/PoweredBy. [Accessed: 2017].

[13] Apache Hadoop. [Online]. Available: https://hadoop.apache.org/.

[14] Apache Flink. [Online]. Available: https://flink.apache.org. [Accessed: 2017].

[15] Apache Spark. [Online]. Available: https://spark.apache.org. [Accessed: 2017].

[16] K. V. Vishwanath and N. Nagappan, “Characterizing Cloud Computing Hardware Reliability,” *Proc. 1st ACM Symp. Cloud Comput. - SoCC ’10*, p. 193, 2010.

[17] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, “An Analysis of Traces from a Production MapReduce Cluster,” in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010, pp. 94–103.

[18] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, “Hadoop’s adolescence,” *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 853–864, 2013.

[19] M. C. Schatz, “CloudBurst: highly sensitive read mapping with MapReduce,” *Bioinformatics*, vol. 25, no. 11, pp. 1363–1369, Jun. 2009.

[20] H. Kocakulak and T. T. Temizel, “A Hadoop solution for ballistic image analysis and recognition,” in *2011 International Conference on High Performance Computing & Simulation*, 2011, pp. 836–842.

[21] J. Moran, C. de la Riva, and J. Tuya, “MRTree: Functional Testing Based on MapReduce’s Execution Behaviour,” in *2014 International Conference on Future Internet of Things and Cloud*, 2014, pp. 379–384.

[22] J. Morán, B. Rivas, C. De Riva, J. Tuya, I. Caballero, and M. Serrano, "Configuration / Infrastructure-aware testing of MapReduce programs," *Adv. Sci. Technol. Eng. Syst. J.*, vol. 2, no. 1, pp. 90–96, 2017.

[23] A. Orso and G. Rothermel, "Software testing: a research travelogue (2000–2014)," *Proc. Futur. Softw. Eng. - FOSE 2014*, pp. 117–132, 2014.

[24] A. Bertolino, "Software Testing Research : Achievements , Challenges , Dreams," in *Future of Software Engineering. FOSE '07*, 2007, pp. 85–103.

[25] S. Nachiyappan and S. Justus, "Getting ready for BigData testing: A practitioner's perception," in *2013 4th International Conference on Computing, Communications and Networking Technologies, ICCCNT 2013*, 2013.

[26] A. Mittal, "Trustworthiness of Big Data," *Int. J. Comput. Appl.*, vol. 80, no. 9, pp. 35–40, Oct. 2013.

[27] Z. Liu, "Research of performance test technology for big data applications," *2014 IEEE Int. Conf. Inf. Autom. ICIA 2014*, no. July, pp. 53–58, 2014.

[28] A. S. Nagdive, R. M. Tugnayat, P. Shri, S. Agnihotri, and M. P. Tembhurkar, "Overview on Performance Testing Approach in Big Data," *Int. J. Adv. Res. Comput. Sci.*, vol. 5, no. 8.

[29] M. Gudipati, S. Rao, N. D. Mohan, and N. Kumar Gajja, "Big Data: Testing Approach to Overcome Quality Challenges," vol. 11, no. 1. Big Data: Challenges and Opportunities, pp. 65–72, 2013.

[30] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, and L. Zhou, "Nondeterminism in MapReduce considered harmful? an empirical study on non-commutative aggregators in MapReduce programs," in *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, 2014, pp. 44–53.

[31] C. Csallner, L. Fegaras, and C. Li, "New Ideas Track: Testing Mapreduce-style Programs," *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, pp. 504–507, 2011.

[32] Y.-F. Chen, C.-D. Hong, N. Sinha, and B.-Y. Wang, "Commutativity of Reducers," Springer Berlin Heidelberg, 2015, pp. 131–146.

[33] L. C. Camargo and S. R. Vergilio, "Classifica{ç} ao de Defeitos para Programas MapReduce: Resultados de um Estudo Emp{\i}rico," 2013.

[34] J. Moran, B. Rivas, C. De La Riva, J. Tuya, I. Caballero, and M. Serrano, "Infrastructure-Aware Functional Testing of MapReduce Programs," in *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, 2016, pp. 171–176.

[35] F. Faghri, S. Bazarbayev, M. Overholt, R. Farivar, R. H. Campbell, and W. H. Sanders, "Failure scenario as a service (FSaaS) for Hadoop clusters," *Proc. Work. Secur. Dependable Middlew. Cloud Monit. Manag. - SDMCMM '12*, pp. 1–6, 2012.

[36] P. Joshi, H. S. Gunawi, and K. Sen, "PREFAIL: A Programmable Tool for Multiple-Failure Injection," *ACM SIGPLAN Not.*, vol. 46, no. 10, p. 171, 2011.

[37] Anarchy Ape. [Online]. Available: https://github.com/david78k/anarchyape.

[38] Chaos Monkey. [Online]. Available: https://github.com/Netflix/SimianArmy.

[39] Hadoop Injection Framework. [Online]. Available: https://hadoop.apache.org/.

[40] J. E. Marynowski, A. O. Santin, and A. R. Pimentel, "Method for testing the fault tolerance of MapReduce frameworks," *Comput. Networks*, vol. 86, pp. 1–13, 2015.

[41] 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 - A-TEST 2015*, 2015, pp. 20–25.

[42] A. J. de Mattos, "Test data generation for testing mapreduce systems," Federal University of Paraná, 2011.

[43] N. Li, Y. Lei, H. R. Khan, J. Liu, and Y. Guo, "Applying combinatorial test data generation to big data applications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, 2016, pp. 637–647.

[44] Herriot: Large-scale automated test framework. [Online]. Available: https://wiki.apache.org/hadoop/HowToUseSystemTestFramework.

[45] Minicluster. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/CLIMiniCluster.html. [Accessed: 27-Feb-2017].

[46] Apache MRUnit. [Online]. Available: http://mrunit.apache.org. [Accessed: 2017].

[47] JUnit. [Online]. Available: http://junit.org. [Accessed: 2017].

[48] L. Baresi and C. Ghezzi, "The Disappearing Boundary Between Development-time and Run-time," *Work. Futur. Softw. Eng. Res. FSE/SDP*, pp. 17–22, 2010.

[49] M. Ali, F. De Angelis, D. Fani, A. Bertolino, G. De Angelis, and A. Polini, "An extensible framework for online testing of choreographed services," *Computer (Long. Beach. Calif).*, vol. 47, no. 2, pp. 23–29, 2014.

[50] E. M. Fredericks, A. J. Ramirez, and B. H. C. Cheng, "Towards run-time testing of dynamic adaptive systems," *2013 8th Int. Symp. Softw. Eng. Adapt. Self-Managing Syst.*, pp. 169–174, 2013.

[51] N. Delgado, A. Q. Gates, and S. Roach, "A Taxonomy and Catalog of Runtime Software- Fault Monitoring Tools," *IEEE Trans. Softw.*, vol. 30, no. 12, pp. 1–16, 2004.

[52] C. Murphy, G. Kaiser, I. Vo, and M. Chu, "Quality Assurance of Software Applications Using the In Vivo Testing Approach," in *2009 International Conference on Software Testing Verification and Validation*, 2009, pp. 111–120.

[53] A. Jacobs, "The Pathologies of Big Data," *Queue*, vol. 7, no. 6, p. 10, 2009.

[54] J. Gao, C. Xie, and C. Tao, "Big Data Validation and Quality Assurance -- Issuses, Challenges, and Needs," in *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2016, no. August, pp. 433–441.

[55] E. J. Weyuker, "On testing non-testable programs," *Comput. J.*, vol. 25, no. 4, pp. 465–470, 1982.

[56] S. Chiba and M. Nishizawa, "An Easy-to-Use Toolkit for Efficient Java Bytecode Translators," Springer Berlin Heidelberg, 2003, pp. 364–376.

[57] R. (MIT L. for C. S. Rivest, "The MD5 Message-Digest Algorithm," *IETF*. pp. 1–22, 1992.

[58] S. Mostafa and X. Wang, "An Empirical Study on the Usage of Mocking Frameworks in Software Testing," in *2014 14th International Conference on Quality Software*, 2014, pp. 127–132.

[59] Mockito. [Online]. Available: http://mockito.org/.

[60] PowerMock. [Online]. Available: http://powermock.github.io/.

[61] HBase. [Online]. Available: https://hbase.apache.org/.

[62] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[63] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, 1985.

[64] SequenceFile. [Online]. Available: https://wiki.apache.org/hadoop/SequenceFile.

[65] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett : Coping with Skewed Content Popularity in MapReduce Clusters," in *Proceedings of the sixth conference on Computer systems-EuroSys '11 (2011)*, 2011, pp. 287–300.

[66] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: a cross-industry study of MapReduce workloads," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1802–1813, 2012.

[67] G. Ananthanarayanan, A. Ghodsi, and A. Wang, "PACMan: Coordinated memory caching for parallel jobs," *9th USENIX Symp. Networked Syst. Des. Implement.*, p. 14 pages, 2012.

[68] Hamcrest. [Online]. Available: http://hamcrest.org/. [Accessed: 02-Mar-2017].

[69] M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "Programs, tests, and oracles: the foundations of testing revisited," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, 2011, p. 391.

[70] R. A. P. Oliveira, U. Kanewala, and P. A. Nardi, "Automated test oracles: State of the art, taxonomies, and trends," *Adv. Comput.*, vol. 95, pp. 113–199, 2015.

[71] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 507–525, 2015.

[72] T. Chen, S. Cheung, and S. Yiu, "Metamorphic testing: a new approach for generating next test cases," *... , Hong Kong Univ. Sci. ...*, pp. 1–11, 1998.

[73] M. Blum and S. Kannan, "Designing programs that check their work," *J. ACM*, vol. 42, no. 1, pp. 269–291, Jan. 1995.

[74] P. E. Ammann and J. C. Knight, "Data diversity: an approach to software fault tolerance," *IEEE Trans. Comput.*, vol. 37, no. 4, pp. 418–425, 1988.

[75] X. Xie, J. W. K. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," in *Journal of Systems and Software*, 2011, vol. 84, no. 4, pp. 544–558.

[76] C. Murphy, K. Shen, and G. Kaiser, "Automatic system testing of programs without test oracles," *Proc. eighteenth Int. Symp. Softw. Test. Anal.*, pp. 189–200, 2009.

[77] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortes, "A Survey on Metamorphic Testing," *IEEE Trans. Softw. Eng.*, vol. 42, no. 9, pp. 805–824, Sep. 2016.

[78] T. Chen, F. Kuo, Y. Liu, and A. Tang, "Metamorphic Testing and Testing with Special Values.," in *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2004.

[79] Jetty server. [Online]. Available: http://www.eclipse.org/jetty/. [Accessed: 2017].

[80] Bi and Chunyue, "Research and application of SQLite embedded database technology," *WSEAS Trans. Comput.*, vol. 8, no. 1, pp. 83–92, 2009.

[81] SQLite. [Online]. Available: https://sqlite.org. [Accessed: 23-Feb-2017].

[82] Open Ankus. [Online]. Available: http://www.openankus.org