Universidad de Oviedo

FUNCTIONAL TESTING TECHNIQUES FOR NEW MASSIVE DATA PROCESSING PARADIGMS

Ph.D. Thesis Dissertation in Computer Engineering

Programa de Doctorado en Informática

Author

Jesús Morán Barbón

Supervisors

Dr. Claudio de la Riva Álvarez
Dr. Javier Tuya González

July 2019

The contact of the author, Jesús Morán is:

- moranjesus@uniovi.es
- Computer Science Department, University of Oviedo, Asturias, 33203, Spain

In addition to the committee in charge of evaluating this dissertation and the two supervisors of the thesis, it has been reviewed by the following researchers:

- Franz Wotawa (Graz University of Technology, Austria)
- Arnaud Gotlieb (Simula Research Laboratory, Norway)

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

RESUMEN DEL CONTENIDO DE TESIS DOCTORAL

1.- Título de la Tesis	
Español/Otro Idioma: Técnicas de prueba funcionales para nuevos paradigmas de procesamiento masivo de datos	Inglés: Functional testing techniques for new massive data processing paradigms

2.- Autor	
Nombre: Jesús Morán Barbón	DNI/Pasaporte/NIE: *******
Programa de Doctorado: Programa de Doctorado en Informática	
Órgano responsable: Centro Internacional de Postgrado	

RESUMEN (en español)

Los programas *Big Data* son aquellos que analizan información utilizando nuevos modelos de procesamiento que superan las limitaciones de la tecnología tradicional en cuanto al volumen, velocidad y variedad de los datos procesados. Entre estos, se destaca *MapReduce* que permite procesar grandes cantidades de datos en una infraestructura distribuida que puede cambiar durante la ejecución debido a los frecuentes fallos en la infraestructura y las optimizaciones. El desarrollador sólo diseña el programa, mientras que la ejecución de su funcionalidad es gestionada por un sistema distribuido, tales como asignación de recursos y el mecanismo de tolerancia a fallos, entre otros. Como consecuencia, un programa puede comportarse diferente en cada ejecución porque se adapta automáticamente a los recursos que estén disponibles en cada momento. Esta ejecución no determinista dificulta las pruebas del software y la depuración, especialmente para aquellos programas *MapReduce* con un diseño complejo. A pesar de que tanto el rendimiento y la funcionalidad son importantes, la mayoría de investigación sobre la calidad de los programas *MapReduce* se centra en rendimiento. Por el contrario, hay pocos estudios sobre funcionalidad a pesar de que varias aplicaciones *MapReduce* fallan con regularidad debido a defectos funcionales. Probar y depurar estos defectos es importante, especialmente cuando los programas *MapReduce* realizan tareas críticas.

Esta tesis tiene como objetivo las pruebas y la depuración de programas *MapReduce* a través de nuevos enfoques para detectar y entender los defectos funcionales causados por un incorrecto diseño del programa. Estos defectos de diseño no sólo dependen de los datos de entrada, sino que unas veces pueden enmascararse y otras producirse por ejecutarse no deterministamente debido a la infraestructura distribuida con optimizaciones automáticas. Para detectar estos defectos, la tesis propone una **técnica de prueba que ejecuta cada caso de prueba con diferentes configuraciones** y que comprueba que esas ejecuciones generen siempre salidas similares. La técnica genera las configuraciones mediante Random testing, y Partition testing junto Combinatorial testing para simular las ejecuciones no deterministas que podrían ocurrir en un entorno de producción. Esta técnica está también automatizada utilizando un motor de ejecución de pruebas que es capaz de detectar estos defectos utilizando sólo los datos de entrada de las pruebas independientemente de su salida esperada.

Una vez que se detecta el defecto, la tesis propone un **framework automático de depuración para localizar la casusa raíz del fallo y aislar los datos que causaron el fallo**. La causa raíz del fallo es automáticamente localizada a través de una técnica spectrum-based que analiza estadísticamente las características que tienen en común aquellas ejecuciones que causan el fallo y aquellas características que lo enmascaran.

Los datos del caso de prueba se reducen para mejorar el entendimiento del fallo utilizando delta debugging y una técnica guiada por búsqueda que iterativamente reducen los datos a la vez que estos nuevos datos siguen causando el fallo. El framework de depuración propuesto en la tesis también permite la inspección de la ejecución distribuida a través de las habituales utilidades de depuraciones como los breakpoints y los watchpoints.

Las anteriores técnicas de pruebas y depuraciones también pueden ser utilizadas en operación. La tesis propone un **enfoque autónomo para detectar los defectos de diseño de los programas *MapReduce* que se están ejecutando en el entorno de producción**. Este enfoque utiliza los datos que se están procesando en tiempo de ejecución en producción como entradas de los casos de prueba para detectar defectos de diseño en operaciones.

Estas técnicas son evaluadas mediante experimentos controlados utilizando programas *MapReduce* reales. Los resultados muestran que las técnicas propuestas son capaces de probar y depurar los programas *MapReduce* automáticamente en pocos segundos. La técnica de prueba detecta la mayoría de los defectos de diseño de los programas *MapReduce*. Una vez que los defectos son detectados, la técnica de localización de defectos localiza habitualmente la causa raíz del defecto, y la técnica de reducción aísla la mayoría de los datos que causan un fallo para mejorar el entendimiento del defecto. En la técnica de reducción de datos, la técnica delta debugging reduce los datos en pocos segundos, por el contrario, el enfoque guiado por búsqueda consume más tiempo pero también reduce más los datos.

Como conclusiones, las técnicas tradicionales de pruebas no son capaces de detectar estos defectos de diseño y las aplicaciones *MapReduce* deben ser probadas con enfoques nuevos como los propuestos en esta tesis. Una vez que el defecto de diseño es detectado, las técnicas de depuración ayudan a entender el fallo, pero las técnicas de depuración tradicionales cubren ampliamente los fallos causados por el código en lugar de aquellos causados por un incorrecto diseño como los de los programas *MapReduce*. Desde el punto de vista de los defectos de diseño funcionales, las aplicaciones *MapReduce* deben ser tanto probadas como depuradas con nuevos enfoques como los que se proponen en esta tesis.

RESUMEN (en Inglés)

Big Data programs are those that analyse the information using new processing models to overcome the limitations of the traditional technology due the volume, velocity or variety of the data. Among them, *MapReduce* stands out by allowing for the processing of large data over a distributed infrastructure that can change during runtime due the frequent infrastructure failures and optimizations. The developer only designs the program, whereas the execution of its functionality is managed by a distributed system, such as the allocation of the resources and the fault tolerance mechanism, among others. As a consequence, a program can behave differently at each execution because it is automatically adapted to the resources available at each moment. This non-deterministic execution makes both software testing and debugging difficult, specially for those *MapReduce* programs with complex design. Despite both performance and functionality are important, the majority of the research about the quality of the *MapReduce* programs are focused on performance. In contrast, few research studies are about functionality although several *MapReduce* applications fail regularly due a functional fault. Testing and debugging these faults are important, specially when the *MapReduce* programs perform a critical task.

This thesis aims to both testing and debugging the *MapReduce* programs with new approaches to detect and understand those functional faults caused by the wrong design of the program. These design faults not only depend on the input data, but they may be triggered sometimes and masked other times because the execution is non-deterministic due a distributed infrastructure with automatic optimizations. To detect these faults, the thesis proposes a **testing technique that executes each test case in different configurations** and then checks that the executions generate always similar outputs. The technique generates the configurations with Random testing, and Partition testing together with Combinatorial testing to simulate the non-determinist executions that could happen in a production environment. This technique is also automated by using a test execution engine that is able to detect these faults using only the test input data, regardless of the expected output.

Once the design faults are detected, the thesis proposes an **automatic debugging framework to locate the root cause of the fault and isolate the data that trigger the failure**. The root cause of the fault is automatically located through a spectrum-based technique that analyses statistically the common characteristic of the executions that trigger the fault against those characteristics that masked the fault. The data of the test case are reduced to improve the fault understanding through delta debugging and search-based techniques that iteratively reduce the data at the same time that the new data still trigger the failure. The debugging framework proposed in the thesis, also allows the inspection of the distributed execution through the common debugging utilities like breakpoints and watchpoints.

The previous testing and debugging techniques can also be used in operations. This thesis proposes an **autonomous approach to detect design faults in *MapReduce* programs executed in production environment**. This approach uses the runtime data as test input data in order to detect the design faults in operations.

These techniques are evaluated through controlled experiments using real-world *MapReduce* programs. The results show that the techniques proposed are able to test and debug the *MapReduce* programs automatically in few seconds. The testing technique detects the majority of the design faults of the *MapReduce* programs. Once the faults are detected, the fault localization technique usually locates the root cause of the design fault, and the reduction technique isolates the majority of the input data that triggers the failure to improve the fault understanding. In the reduction technique, the delta debugging technique reduces the data in a few seconds, in contrast the search-based approach is more time consuming but also reduces more data.

As conclusions, the traditional testing techniques are not able to detect these design faults and the *MapReduce* applications must be tested with new approaches like those proposed in this thesis. Once the design fault is detected, the debugging techniques help to understand this fault, but the traditional debugging techniques are broadly focused on the failures caused by the code instead of those caused by a wrong design like in the *MapReduce* programs. From the point of view of functional design faults, the *MapReduce* applications must be both tested and debugged with new approaches like those proposed in this thesis.

SR. PRESIDENTE DE LA COMISIÓN ACADÉMICA DEL PROGRAMA DE DOCTORADO EN INFORMÁTICA

CONTENTS

In memoriam of my grandmother

ACKNOWLEDGEMENTS

First and foremost, I would like to express my special thanks to my supervisors Claudio de la Riva and Javier Tuya for all continuous support and dedication during the years of the thesis degree not only in the research, but also in academic and my career. I am grateful to my other colleagues of the research group GIIS (Software Engineering Research Group) who gave me the opportunity to discuss the research contributions and help me to improve my research skills.

Besides the researchers of my university, I also would like to thanks to the SEDC research group (Software Engineering and Dependable Computing Laboratory) to gave me the opportunity to access their laboratory at Pisa (Italy). My humble thanks to Antonia Bertolino for all valuable guidance in our collaborations that improved my skills, and also to provide me the opportunity to work together.

My special thanks to my family for provide me both education and support along my whole life. Among all of my family members, I want to highlight my mother, father, brother together with his wife, and my nieces.

Last but not the least, I would like to thank Laura for all advises provided me during the last years that makes me grow personally, and also to for being always there whenever I needed her.

This work has been performed under the research projects TIN2013-46928-C3-1-R, funded by the Spanish Ministry of Economy and Competitiveness; GRUPIN14-007, funded by the Principality of Asturias (Spain); TIN2016-76956-C3-1-R, funded by the Spanish Ministry of Science and Technology; the Severo Ochoa pre-doctoral grant BP16215; and ERDF Funds.

ABSTRACT

Big Data programs are those that analyse the information using new processing models to overcome the limitations of the traditional technology due the volume, velocity or variety of the data. Among them, *MapReduce* stands out by allowing for the processing of large data over a distributed infrastructure that can change during runtime due the frequent infrastructure failures and optimizations. The developer only designs the program, whereas the execution of its functionality is managed by a distributed system, such as the allocation of the resources and the fault tolerance mechanism, among others. As a consequence, a program can behave differently at each execution because it is automatically adapted to the resources available at each moment. This non-deterministic execution makes both software testing and debugging difficult, specially for those *MapReduce* programs with complex design. Despite both performance and functionality are important, the majority of the research about the quality of the *MapReduce* programs are focused on performance. In contrast, few research studies are about functionality although several *MapReduce* applications fail regularly due a functional fault. Testing and debugging these faults are important, specially when the *MapReduce* programs perform a critical task.

This thesis aims to both testing and debugging the *MapReduce* programs with new approaches to detect and understand those functional faults caused by the wrong design of the program. These design faults not only depend on the input data, but they may be triggered sometimes and masked other times because the execution is non-deterministic due a distributed infrastructure with automatic optimizations. To detect these faults, the thesis proposes a **testing technique that executes each test case in different configurations** and then checks that the executions generate always similar outputs. The technique generates the configurations with Random testing, and Partition testing together with Combinatorial testing to simulate the non-determinist executions that could happen in a production environment. This technique is also automated by using a test execution engine that is able to detect these faults using only the test input data, regardless of the expected output.

Once the design faults are detected, the thesis proposes an **automatic debugging framework to locate the root cause of the fault and isolate the data that trigger the failure**. The root cause of the fault is automatically located through a spectrum-based technique that analyses statistically the common characteristic of the executions that trigger the fault against those characteristics that masked the fault. The data of the test case are reduced to improve the fault understanding through delta debugging and search-based techniques that iteratively reduce the data at the same time that the new data still trigger the failure. The debugging framework proposed in the thesis, also allows the inspection of the distributed execution through the common debugging utilities like breakpoints and watchpoints.

The previous testing and debugging techniques can also be used in operations. This thesis proposes an **autonomous approach to detect design faults in *MapReduce* programs executed in production environment**. This approach uses the runtime data as test input data in order to detect the design faults in operations.

These techniques are evaluated through controlled experiments using real-world *MapReduce* programs. The results show that the techniques proposed are able to test and debug the *MapReduce* programs automatically in few seconds. The testing technique detects the majority of the design faults of the *MapReduce* programs. Once the faults are detected, the fault

localization technique usually locates the root cause of the design fault, and the reduction technique isolates the majority of the input data that triggers the failure to improve the fault understanding. In the reduction technique, the delta debugging technique reduces the data in a few seconds, in contrast the search-based approach is more time consuming but also reduces more data.

As conclusions, the traditional testing techniques are not able to detect these design faults and the *MapReduce* applications must be tested with new approaches like those proposed in this thesis. Once the design fault is detected, the debugging techniques help to understand this fault, but the traditional debugging techniques are broadly focused on the failures caused by the code instead of those caused by a wrong design like in the *MapReduce* programs. From the point of view of functional design faults, the *MapReduce* applications must be both tested and debugged with new approaches like those proposed in this thesis.

RESUMEN

Los programas *Big Data* son aquellos que analizan información utilizando nuevos modelos de procesamiento que superan las limitaciones de la tecnología tradicional en cuanto al volumen, velocidad y variedad de los datos procesados. Entre estos, se destaca *MapReduce* que permite procesar grandes cantidades de datos en una infraestructura distribuida que puede cambiar durante la ejecución debido a los frecuentes fallos en la infraestructura y las optimizaciones. El desarrollador sólo diseña el programa, mientras que la ejecución de su funcionalidad es gestionada por un sistema distribuido, tales como asignación de recursos y el mecanismo de tolerancia a fallos, entre otros. Como consecuencia, un programa puede comportarse diferente en cada ejecución porque se adapta automáticamente a los recursos que estén disponibles en cada momento. Esta ejecución no determinista dificulta las pruebas del software y la depuración, especialmente para aquellos programas *MapReduce* con un diseño complejo. A pesar de que tanto el rendimiento y la funcionalidad son importantes, la mayoría de investigación sobre la calidad de los programas *MapReduce* se centra en rendimiento. Por el contrario, hay pocos estudios sobre funcionalidad a pesar de que varias aplicaciones *MapReduce* fallan con regularidad debido a defectos funcionales. Probar y depurar estos defectos es importante, especialmente cuando los programas *MapReduce* realizan tareas críticas.

Esta tesis tiene como objetivo las pruebas y la depuración de programas *MapReduce* a través de nuevos enfoques para detectar y entender los defectos funcionales causados por un incorrecto diseño del programa. Estos defectos de diseño no sólo dependen de los datos de entrada, sino que unas veces pueden enmascararse y otras producirse por ejecutarse no deterministamente debido a la infraestructura distribuida con optimizaciones automáticas. Para detectar estos defectos, la tesis propone una **técnica de prueba que ejecuta cada caso de prueba con diferentes configuraciones** y que comprueba que esas ejecuciones generen siempre salidas similares. La técnica genera las configuraciones mediante Random testing, y Partition testing junto Combinatorial testing para simular las ejecuciones no deterministas que podrían ocurrir en un entorno de producción. Esta técnica está también automatizada utilizando un motor de ejecución de pruebas que es capaz de detectar estos defectos utilizando sólo los datos de entrada de las pruebas independientemente de su salida esperada.

Una vez que se detecta el defecto, la tesis propone un **framework automático de depuración para localizar la casusa raíz del fallo y aislar los datos que causaron el fallo**. La causa raíz del fallo es automáticamente localizada a través de una técnica spectrum-based que analiza estadísticamente las características que tienen en común aquellas ejecuciones que causan el fallo y aquellas características que lo enmascaran. Los datos del caso de prueba se reducen para mejorar el entendimiento del fallo utilizando delta debugging y una técnica guiada por búsqueda que iterativamente reducen los datos a la vez que estos nuevos datos siguen causando el fallo. El framework de depuración propuesto en la tesis también permite la inspección de la ejecución distribuida a través de las habituales utilidades de depuraciones como los breakpoints y los watchpoints.

Las anteriores técnicas de pruebas y depuraciones también pueden ser utilizadas en operación. La tesis propone un **enfoque autónomo para detectar los defectos de diseño de los programas *MapReduce* que se están ejecutando en el entorno de producción**. Este enfoque utiliza los datos que se están procesando en tiempo de ejecución en producción como entradas de los casos de prueba para detectar defectos de diseño en operaciones.

Estas técnicas son evaluadas mediante experimentos controlados utilizando programas *MapReduce* reales. Los resultados muestran que las técnicas propuestas son capaces de probar y depurar los programas *MapReduce* automáticamente en pocos segundos. La técnica de prueba detecta la mayoría de los defectos de diseño de los programas *MapReduce*. Una vez que los defectos son detectados, la técnica de localización de defectos localiza habitualmente la causa raíz del defecto, y la técnica de reducción aísla la mayoría de los datos que causan un fallo para mejorar el entendimiento del defecto. En la técnica de reducción de datos, la técnica delta debugging reduce los datos en pocos segundos, por el contrario, el enfoque guiado por búsqueda consume más tiempo pero también reduce más los datos.

Como conclusiones, las técnicas tradicionales de pruebas no son capaces de detectar estos defectos de diseño y las aplicaciones *MapReduce* deben ser probadas con enfoques nuevos como los propuestos en esta tesis. Una vez que el defecto de diseño es detectado, las técnicas de depuración ayudan a entender el fallo, pero las técnicas de depuración tradicionales cubren ampliamente los fallos causados por el código en lugar de aquellos causados por un incorrecto diseño como los de los programas *MapReduce*. Desde el punto de vista de los defectos de diseño funcionales, las aplicaciones *MapReduce* deben ser tanto probadas como depuradas con nuevos enfoques como los que se proponen en esta tesis.

I INTRODUCTION

I.1 RESEARCH CONTEXT

In the last decades the volume of data generated by companies has grown exponentially expecting to increase by 300 times from 2005 to 2020 [1]. This grow, not only in size, but also in complexity, arise several challenges to storing, transporting and analysing such information. To overcome these challenges, novel technologies are being created under the *Big Data* paradigm [2]. Their rise allows large scale analysis of data, from social web interactions to industrial sensor data, that can improve social and business performance.

The adoption of and interest in these technologies/paradigms has increased over the last few years to the extent that several Fortune 1000 enterprises consider *Big Data* critical for business [3]. Despite the importance of these applications, some studies predicted that 60% of *Big Data* projects fail to go beyond piloting and would be abandoned in 2017 [4]. There are several obstacles and challenges in this paradigm. *Big Data* involves at the same time several fields such as data science to analyse the data, information technology to make it possible in a scalable way through new tools and technology beyond the state-of-the-art, and business science to find value from the analysis. These transversal skills are difficult to find in *Big Data* and in future years a shortage of experts is expected [5]. The lack of skills is also currently one of the main concerns [6]–[8] and causes that several companies are able to capture the data but not to process them [9]. Another obstacle is the poor data quality [10] that it is among the main concerns in *Big Data* [6]. Dealing with these complex data to integrate and process them is a challenge [11] and could incur in large costs. In U.S. economy the bad data cost around 3.1$ trillion per year [12] together with other derivate problems as for example the 6.8 billion of email pieces that could not be sent during 2013 by USPS (U.S. Postal Service) due to data quality issues among other problems in the address information [13]. The previous concerns and challenges, among other technological issues [7], [14], [15], can lead the *Big Data* projects to failures that impact both business and society.

The *MapReduce* processing model [16] stands out among *Big Data* applications. It is a key technology very broadly used by organizations [17] and implemented in several mature frameworks [18], [19], such as Hadoop [20], Flink [21], [22] and Spark [23], [24], among others. Because it is so widely adopted, the quality of *MapReduce* programs is important, especially for those employed in critical sectors as such as health (DNA alignment [25]) and security (image processing in ballistics [26]). An analysis over several months at Yahoo! indicates that around 3% of *MapReduce* programs are not finished [27], whereas another broader study places this percentage between 1.38% and 33.11% [28]. A study of 507 programs in production reveals at least 5 different kinds of faults [29] that are caused by the incorrect design of *MapReduce* programs. Therefore, this thesis is focused to address these functional faults that are caused by incorrect design.

These types of faults include, but are not limited to, race conditions, computations with unavailable data because the distributed system allocates them to another computer, or automatic optimizations that remove data that are relevant to calculating the output. These faults are difficult to detect because they depend not only on the data, but also on how these data are executed in the large distributed architecture: parallel executions, re-executions of some part of the data and optimizations, among others. In general, these non-deterministic faults are prone to be masked in development/testing environments and go on to fail in more aggressive environments such as the production environment, thereby generating incorrect outputs or causing the program to crash. The distributed nature of these design faults complicates also the debugging. The localization of the root cause of the fault is complex because not only involves the code, but the design of the program. Then the developer could have difficulties to both understand and fix these non-deterministic design faults.

There are several approaches to assess quality, and software testing is one of the most commonly used. According to the ISO/IEC/IEEE 29119-1:2013 standard [30], software testing aims to provide information relating to program quality and the potential impacts/risks of poor quality. Software testing research has evolved in recent years [31], but there are several challenges related to the testing of programs in cloud and adaptive architectures [32].

I.2 Research Hypothesis

This thesis focuses on the quality of the *MapReduce* programs, especially in testing and debugging. The research performed is based in the following hypothesis:

H1. The *MapReduce* applications have specific characteristics that another kind of applications do not have, such as delegate their execution to a framework that handles the massive execution splitting the datasets along several servers, allocating resources in parallel, or re-executing of part of the program in case of infrastructure failures. These characteristics in conjunction are not broadly covered by the state-of-the-art testing techniques, and the *MapReduce* applications must be tested with new approaches.

H2. The functional failures of the *MapReduce* applications that are wrongly designed entail the execution of the data concurrently in several servers in non-deterministic way. These failures are not just caused by the code, but by the design. The traditional debugging techniques are broadly focused on the failures caused by the code but not on those caused by the wrong design, then the *MapReduce* applications must be debugged with new approaches.

I.3 Research Goals

The general goal of this thesis is to enhance the state-of-the-art of the software testing and debugging in the *MapReduce* field with experimentation and pragmatic basis. This goal is divided in the following sub-goals:

1. Determine the state-of-the-art and the current quality problems of the *Big Data* programs that are executed through the *MapReduce* processing model.
2. Develop new testing techniques to address the design of *MapReduce* applications that are executed in scalable way under both distributed and heterogeneous large infrastructures.
3. Design new debugging techniques that provides an easy understanding of the complex faults of distributed *MapReduce* applications.
4. Integrate and automate both the testing and debugging activities in a general quality process that can be applied in laboratory (pre-production) or during runtime monitoring the operations (production).
5. Evaluate the effectiveness and efficiency of the testing and debugging techniques proposed.

I.4 Research Methodology

To achieve the previous goals, the thesis was planned according to the below research and engineering methodologies/practices:

- Systematic Literature Review [33] (systematic mapping study [34]): This kind of studies are focused to extract evidences through the analysis of the relevant literature. In Chapter II, the state-of-the-art of the software testing in the *MapReduce* applications is defined through the revision and classification of the literature applying a strict protocol to avoid bias.
- Action research [35]: This methods pursue to join the scientific theory with the industrial practice through reflection feedback. In this thesis, the academic research is performed and discussed together with industrial organizations in order to frame their real problems. Some programs used to validate the research comes from the industry.
- Experimentation: The contributions of the thesis are validated with experiments [36] using the usual techniques of software engineering field through real-world programs.

I.5 Contributions

The main contribution of the thesis is the devise and automatization of both testing and debugging techniques in the *MapReduce* field. This main contribution is characterized in the following contributions:

C1. Analysis of the state-of-the-art, challenges and gaps of software testing in the *MapReduce* application by means of systematic mapping study of the literature evidences.

C2. Testing technique to automatically detects the design faults of the *MapReduce* application using input partitioning, combinatorial and random techniques, and metamorphic testing.

C3. Debugging technique to automatically localize the root cause of the design faults in *MapReduce* applications through spectrum-based fault localization approach.

C4. Debugging technique to automatically isolate the data that triggers the design faults of the *MapReduce* application using Delta debugging and a search-based technique based on genetic algorithm.

C5. Automatization and integration of both testing and debugging techniques in a tool that support offline testing in laboratory.

C6. Automatization and integration of both testing and debugging techniques in an autonomous system that performs testing-debugging without human intervention in production taking real data at runtime.

C7. Experimentation with real-world programs of both testing and debugging techniques.

These contributions are tackled in four nested lines of research. The first line is focused on the state-of-the-art to hypothesize about quality problems of *MapReduce* programs and how can be alleviated through research in both testing and debugging fields (C1). Then the second and third lines of research aboard testing (C2, C5, C7) and debugging (C3, C4, C7), respectively. Finally, the fourth line of research is focused on the integration of our work to be used not only in laboratory, but also on production in autonomous way (C6).

Despite the four line of research are mature with several publications, the ongoing work of the debugging is not yet published. An early version of the debugging techniques is published in two national conferences, but the last version is finished aimed to be published in a JCR journal. The remainder contributions are fully published in both journals and conferences, and they are also divulgated at different venues as Fig. 1 summarizes.

This work was done in collaboration with the Institute of the National Research Council of Italy (CNR-ISTI), and additionally with the University of Castilla-La Mancha of Spain (UCLM). These collaborations were done through periodical videoconferences and visits. Morán visited UCLM one week, and CNR-ISTI during two internships of three months each one.

The contributions of the thesis were also divulgated in different forums through posters and talks to discuss the vision of the research goals with another researchers that provides feedback. As a result, several tools were developed to support and automatize the research in all processes.

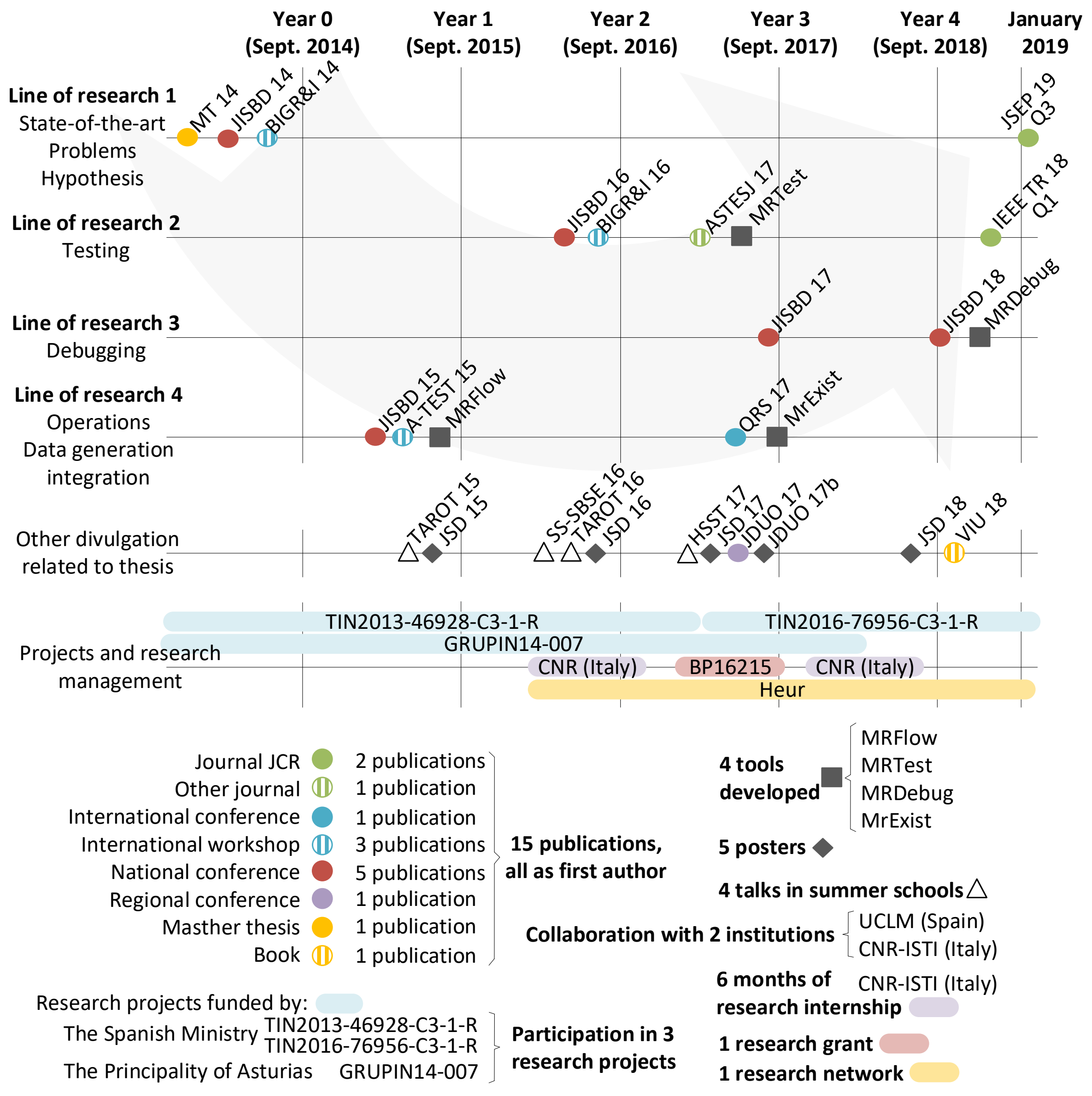

Fig. 1 Summary of the thesis research and divulgation

I.5.1 First line of research: State-of-the-art, problems and hypothesis

The first line of research has the following 4 publications:

- JSEP 2019 [37] (Q3): **Morán, J.**, de la Riva, C., Tuya, J. "Testing MapReduce programs: A systematic Mapping Study". Journal of Software: Evolution and Process, 2019.
- BIGR&I 2014 [38]: **Morán, J.**, de la Riva, C., Tuya, J. "MRTree: Functional testing based on MapReduce's execution behaviour". International Symposium on Big Data Research and Innovation, 380-384, 2014.
- JISBD 2014 [39]: **Morán, J.**, de la Riva, C., Tuya, J. "Pruebas funcionales en programas MapReduce basadas en comportamientos no esperados". Jornadas de Ingeniería del Software y Bases de Datos, 2014.
- MT 2014 [40] (industrial prize): **Morán, J.**, de la Riva, C. "Pruebas para sistemas con procesamiento y transformación de datos en paralelo". Master thesis dissertation, University of Oviedo, 2014.

We revised rigorously the literature about software testing in *MapReduce* applications summarizing the state-of-the-art, problems and gaps through systematic mapping study [37]. Among other findings, we observed gaps in the functional characteristic of the program quality. There are evidences that both performance and functional characteristics are relevant in *MapReduce* programs, but the majority of efforts are focused on performance and few in functional point of view. Then this thesis is focused to embrace this gap providing evidences, techniques, results and tools that mitigate the functional faults of *MapReduce* programs through testing and debugging.

The general testing techniques can detect some faults of the *MapReduce* applications, but are unsuitable to detect those other faults that are specific of the processing model. Some of these faults are caused by the wrong design of the *MapReduce* application. The developer designs the program at high level using some functionalities such as Map, Reduce or Combine. Then the execution of the program is automatically managed by the framework, typically Hadoop, Spark or Flink. The programs must be designed accordingly to support all possibilities proposed at runtime by the framework and the environment, such as different parallelization degrees, re-execution of some parts due infrastructure failures, or optimizations to avoid net bottlenecks removing some irrelevant data. We identified and classified these design faults [38], [39] aimed to approach them through testing in the second research line.

With this classification of design faults, we published a master thesis [40] that, despite is focused on research, received an industrial prize.

I.5.2 Second line of research: Testing

The second line of research is focused on these design faults through testing and has the following 4 publications:

- IEEE TR 2018 [41] (Q1): **Morán, J.**, Bertolino, A., de la Riva, C., Tuya, J. "Automatic Testing of Design Faults in MapReduce Applications". IEEE Transactions on Reliability, vol. 67, no. 3, pp. 717-732, 2018.
- ASTESJ 2017 [42]: **Morán, J.**, Rivas, B., Riva, C., Tuya, J, Caballero, I., Serrano. Configuration/Infrastructure-aware testing of MapReduce programs. Advances in Science, Technology and Engineering Systems Journal, vol. 2, no. 1, pp. 90-96, 2017.

- BIGR&I 2016 [43]: **Morán, J.**, Rivas, B., de la Riva, C., Tuya, J, Caballero, I., Serrano, M. "Infrastructure-Aware Functional Testing of MapReduce programs". 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW), Vienna, 2016, pp. 171-176.
- JISBD 2016 [44]: **Morán, J.**, de la Riva, C., Tuya, J. "Generación y Ejecución de Escenarios de Prueba para Aplicaciones MapReduce". Jornadas de Ingeniería del Software y Bases de Datos, 2016.

The design faults of the *MapReduce* programs are difficult to test because the same data could yield sometimes success and other failures depending on the execution (parallelism, optimizations and others). This execution is very difficult to control and the faults trend to be masked during laboratory tests because the infrastructure is not as aggressive as the production environment. At first point we explored different ways to execute the data in thoroughly possible situations to check if the program fails or not regardless of the execution [42]–[44]. Finally, we devised a testing technique that automatically detect automatically the design faults given only the data under test [41]. This technique does not execute the program under thoroughly situations, but on those infrastructure configurations that are relevant using input space partitioning, random and combinatorial testing. Then the faults are automatically detected by means of metamorphic testing [45] when the same data do not yield to equivalents outputs in the different infrastructure configurations. As a result, a tool called MRTest was developed:

- MRTest: a xUnit testing tool that detects automatically the design faults of the *MapReduce* applications simulating the controlled execution of different infrastructure configurations.

With this technique and tool, the design faults of the *MapReduce* programs can be automatically detected. These faults are manifested sometimes and masked in others in non-deterministic way. Then the root cause of the fault is difficult to both localize and understand. The common debugging techniques are not suitable for these faults because are focused on code and not neither on infrastructure configurations nor design level.

I.5.3 Third line of research: Debugging

The third line of research is focused on debugging these design fault and has the following 2 publications:

- JISBD 2018 [46]: **Morán, J.**, de la Riva, C., Tuya, J. "Automatización de la localización de defectos en el diseño de aplicaciones MapReduce". Jornadas de Ingeniería del Software y Bases de Datos, 2018.
- JISBD 2017 [47]: **Morán, J.**, de la Riva, C., Tuya, J., Rivas, B. "Localización de defectos en aplicaciones MapReduce". Jornadas de Ingeniería del Software y Bases de Datos, 2017.

The root cause of the design faults can be obtained analysing the characteristics that have in common both the infrastructure configurations that produce a failure and those that success. An initial approach to obtain the root cause of the faults is published with a spectrum-based fault localization technique [46], [47]. The last version of this technique including the controlled experiment is not yet published.

The previous technique is able to localize automatically the characteristics of the infrastructure configurations that produces the failure. However, the faults are still difficult to understand due the distributed flow of the data. In order to improve the understanding of the faults, we devised a technique that isolates the data that are relevant to trigger the fault removing the remainder of the data. This technique uses a search-based approach with delta debugging [48], [49] and a genetic algorithm [50], [51]. This technique is not yet published.

To automatize the localization of the faults and the isolation of the data, we developed a tool that also support the other typical debugging commodities such breakpoints and watchpoints:

- MRDebug: a framework to debug the *MapReduce* programs focused on design faults. The frameworks automatically localizes the root cause of the faults and reduces the data isolating those that are relevant to trigger the failure, then the developer can use breakpoints, watchpoints and the common debugging toolkit.

The majority of this research line is not yet published. We are writing a paper that includes all of our debugging research with controlled experiments aimed to be published in JCR journal.

Both, the testing and debugging, are done automatically based only on the test input data. Then the techniques can be used in production taking advantage of the production data.

I.5.4 Fourth line of research: Operations

The fourth line of research is focused on the operations and generation/selection of the data that can be used during the testing and debugging of the *MapReduce* programs. This research line has the following 3 publications:

- QRS 2017 [52]: **Morán, J.**, Bertolino, A., de la Riva, C., Tuya, J. "Towards Ex Vivo Testing of MapReduce Applications," 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, 2017, pp. 73-80.
- A-TEST 2015 [53]: **Morán, J.**, de la Riva, C., Tuya, J. “Testing data transformations in MapReduce programs”. 6th International Workshop on Automating Test Case Design, Selection and Evaluation, 20-25, 2015.
- JISBD 2015 [54]: **Morán, J.**, de la Riva, C., Tuya, J. “Pruebas basadas en flujo de datos para programas MapReduce”. Jornadas de Ingeniería del Software y Bases de Datos, 2015.

The data of the test cases can be generated manually using a general-purpose testing technique. We adapted the data flow testing technique [55] to the *MapReduce* processing model considering the transformations of the data in the infrastructure configurations [53], [54]. This technique is automatized in a tool called MRFlow:

- MRFlow: a tool that indicates the test coverage items to test the programs based on data-flow testing technique adapted to *MapReduce*.

The testing and debugging techniques only need data to be executed, and in *Big Data* field there are a lot of data. Then these techniques can take advantage of the runtime data. It is not possible to perform testing/debugging with all huge production data, and we devise a technique to extract and cache samples of the data to the laboratory at runtime. Then when a user executes a program in production, the tests of the program are performed in the laboratory in autonomous way without human intervention. As a result, a system called MrExist was developed:

- MrExist: an autonomous system that detects the execution of a *MapReduce* program in the production cluster and performs automatically testing/debugging in laboratory taking data from runtime.

I.5.5 Other Divulgation related to the Thesis

The research topics of the thesis were divulgated and discussed with other researchers at the following venues:

Doctoral workshop:

- JDUO 2017: **Morán, J.** "Técnicas de Prueba Funcionales para Nuevos Paradigmas de Procesamiento Masivo de Datos". VI Jornadas Doctorales de la Universidad de Oviedo, 2017.

Posters:

- JSD 2018: **Morán, J.**, de la Riva, C., Tuya, J. "Functional testing techniques for new massive data processing paradigms". Jornada seguimiento doctorado. Universidad de Oviedo, 2018.
- JDUO 2017b: **Morán, J.** "Functional testing techniques for new massive data processing paradigms". Poster VI Jornadas Doctorales de la Universidad de Oviedo, 2017.
- JSD 2017: **Morán, J.**, de la Riva, C., Tuya, J. "Functional testing techniques for new massive data processing paradigms". Jornada seguimiento doctorado. Universidad de Oviedo, 2017.
- JSD 2016: **Morán, J.**, de la Riva, C., Tuya, J. "Functional testing techniques for new massive data processing paradigms". Jornada seguimiento doctorado. Universidad de Oviedo, 2016.
- JSD 2015: **Morán, J.**, de la Riva, C., Tuya, J. "Functional testing techniques for new massive data processing paradigms". Jornada seguimiento doctorado. Universidad de Oviedo, 2015.

Talks in summer schools:

- HSST 2017: **Morán, J.** "Automatic Functional Testing of MapReduce Applications". 7th Halmstad Summer School on Testing, 2017.
- TAROT 2016: **Morán, J.** "Software Testing in MapReduce applications". 11th International Summer School on Training And Research On Testing, 2016.
- SS-SBSE 2016: **Morán, J.** "Functional testing of Big Data programs using a combinatorial algorithm". 1st International Summer School on Search-Based Software Engineering, 2016.
- TAROT 2015: **Morán, J.** "Functional Testing of MapReduce programs". 11th International Summer School on Training And Research On Testing, 2015.

Some of the divulgation were about the general lines of the thesis research, but in other cases were specific ongoing work to obtain feedback from the research community.

The thesis is focused on the detection and understanding of the design faults in *MapReduce* programs. This approach is reactive because analyse a program already implemented, but the design faults can also be avoided in proactive way during implementation. Then Morán published a book to explain, in practical way, different design patters that can be used to implement *MapReduce* programs aimed to foster good practices during the design.

Book:

- VIU 2018: **Morán, J.** "Métodos para extracción, procesamiento y almacenamiento de datos masivos desde Internet". Valencia International University, 2018.

I.5.6 Projects and Research Management

The work of this thesis is not done only by one person, but by the collaboration of the University of Oviedo with other 2 organizations: mainly with Institute of the National Research Council of Italy (CNR-ISTI), and additionally with University of Castilla-La Mancha (UCLM). The UCLM collaborates in the initial steps of the testing and debugging techniques, whereas the CNR-ISTI collaborates in the controlled experiments of testing/debugging in both laboratory and production environments.

The collaborations were coordinated through frequent videoconferences, visits and internships. Morán visited UCLM during a week and also met the team members each year in national conferences. During this visit, the collaborators discussed different approaches of the testing technique to be designed, and how to tackle the functional faults of the *MapReduce* programs.

In the collaboration with CNR-ISTI, Morán visited Italy twice in internships of three months each one. During the first internship, the collaborators designed and validated the testing technique proposing different approaches. In the second internship, the debugging technique of localization was mature, and the data isolation were devised with different approaches.

Both collaborations, UCLM and CNR-ISTI, continue in this research lines through videoconferences and future visits/internships.

Internships and collaborations:

- CNR-ISTI (Italy): Lines of research 2, 3 and 4 (IEEE TR 2018 [41] and QRS 2017 [52]).
- UCLM (Spain): Lines of research 2 and 3 (ASTESJ 2017 [42], BIGR&I 2016 [43] and JISBD 2017 [47]).

Other collaborations are done through the specialized research networks. During this thesis, Morán become a member of the HEUR research network that is focused on meta-heuristics.

Research network:

- HEUR: network about meta-heuristics and used in the third line of research.

During the thesis, Morán was member of three research projects aligned with the topics of the research, two leaded by Tuya and other leaded by De la Riva.

Research projects:

- TIN2013-46928-C3-1-R: Research project funded by the Spanish ministry about software testing in Big Data.
- TIN2016-76956-C3-1-R: Research project funded by the Spanish ministry about software testing in both Big Data and mobile.
- GRUPIN14-007: Research project funded by the Principality of Asturias about software testing.

This thesis was also supported by a pre-doctoral grant.

Grant:

- BP16215: Severo Ochoa pre-doctoral grant.

I.6 THESIS ORGANIZATION

The lines of research and organization of the chapters are summarized in Fig. 2. This first chapter is about the thesis research plan. The second chapter defines the state-of-the-art and the problem statement about functional design faults in the *MapReduce* programs (first line of research). Then the third chapter describes how to detect these faults through software testing technique (second line of research). In the fourth chapter, debugging techniques are detailed to localize and understand the root cause of the faults (third line of research). Then the fifth chapter particularized both the testing and debugging techniques to be executed autonomously during operations (production) taking advantage to the runtime data (fourth line of research). The conclusions and future works are summarized in the sixth chapter. Finally, the seventh chapter contains the appendices of the thesis.

This thesis is structured with the following chapters:

Chapter I **"Introduction"**: Contains the introduction of the thesis including the research context, hypothesis, goals, methodologies and contributions.

Chapter II **"Background"**: Defines the state-of-the-art by means of systematic mapping study aimed to find evidences in the literature of software testing in the *MapReduce* programs. This chapter also defines the problematic in which this thesis is focused, that is the design faults of the *MapReduce* programs.

Chapter III **"Testing"**: Describes the testing technique defined to detect automatically the design faults of the *MapReduce* programs. This technique executes the same program simulating different environments generated through input space partitioning, combinatorial and random testing. Then the outputs of these different executions are analysed with metamorphic testing to detect the faults. This testing technique is validated with controlled experiments in real-world programs.

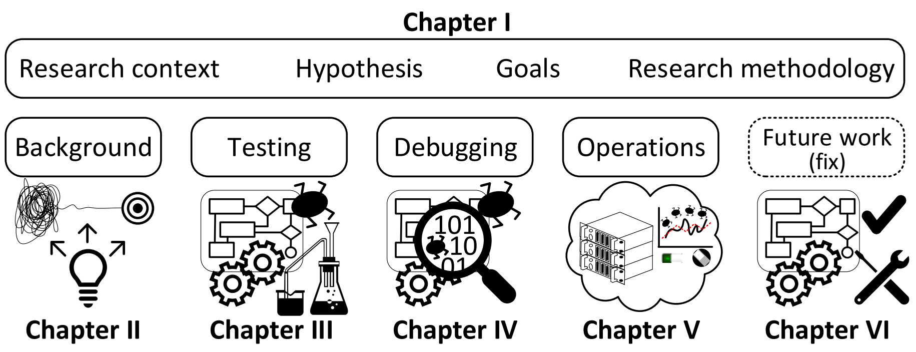

Fig. 2 Summary of the thesis lines of research and organization

Chapter IV **“Debugging”**: Details the debugging techniques defined to both localize and isolate the data that trigger the failure aimed to enhance the understanding of the fault. The majority of debugging techniques are focused on the code, but these design faults are caused by the structuration and not by the code. Then the debugging techniques are adapted to focus on the design by means of (1) spectrum-based fault localization to localize automatically the root cause of the fault in the design, and (2) delta debugging and search-based algorithm to both isolate and reduce the data that trigger the design failure. These debugging techniques are integrated in a framework that also support the common debugging toolkits. The techniques are validated with controlled experiments in real-world programs.

Chapter V **“Operations”**: Particularized the previous testing and debugging techniques to be used autonomously during the operations taking runtime data as test data. The previous testing and debugging techniques can be executed automatically providing only the input data of the test cases. In the field of Big Data, there are a lot of data in production. Then these techniques can be executed in autonomous way taking directly the data from production to detect design faults during runtime. This technique is defined with Ex Vivo approach that consist to take the data from production environment, but executing the tests outside in laboratory to inquire the quality of the program that is executed in production.

Chapter VI **“Final remarks”**: Summarizes the conclusions of the thesis and the future work by means of new research line focused not only to detect the faults automatically during runtime, but to fix the programs with self-adaptation approach and PDCA methodology.

Chapter VII **“Appendices”**: Contains the appendices of the thesis.

II BACKGROUND

This chapter summarizes the broad related work analysing the state-of-the-art of software testing for *MapReduce* programs through a systematic mapping study. The majority of this chapter is published in JSEP 2019 [37]. Section II.1 introduces MapReduce together with the main challenges from the testing point of view. The research questions are proposed in Section II.2 together with the systematic steps planned to answer them. The execution of these steps (conducting) is described in Section II.3. The answers to research questions and other results are detailed in Section II.4. These results are discussed in Section II.5. The confidence of the results obtained from both planning and conducting is enumerated in Section II.6. Finally, Section II.7 contains a summary.

II.1 MAPREDUCE PROCESSING MODEL

MapReduce programs [16] divide one problem into several subproblems that are executed in parallel over a large number of computers. The programs have two principal functions: (1) *Map*, that analyses parts of the input data and classifies them into subproblems, and (2) *Reduce*, that solves each of these subproblems. The data processed by these functions are handled internally in the form of <key, value> pairs. The "key" is the identifier of each subproblem and the value contains information that the subproblem needs to solve. To illustrate *MapReduce*, let us imagine a program that calculates the average temperature per year. This problem could be divided into one subproblem per year, then each subproblem only solves the average temperature in one year. In this program, the "key" is the year because it identifies each subproblem, whereas the "value" is the temperature of this year because this information is needed to solve the subproblem. Fig. 3 details a distributed execution of the program analyzing the years 2000-2003. Firstly, the *Map* function receives the data pertaining to years and temperatures and creates the <key, value> pairs with <year, temperature>. For example, <2000, 3°> means that 3° is needed to solve the subproblem that calculates the average temperature of 2000. Then the *Reduce* function receives from all *Maps* one year with all of its temperatures, and calculates the average. For example, if one *Reduce* function receives the data that in the year 2000 there were 3° and 1° temperatures, that is <2000, [3°, 1°]>, then the average temperature for the year is 2°.

The programs are executed by a framework that automatically manages the resource allocation, the re-execution of one part of the program in case of infrastructure failures, and the scheduling of all executions, among other mechanisms. The data analyzed could be stored in several distributed sources, such as non-relational databases and distributed file systems.

The integration of all of these technologies in the *MapReduce* program stack presents a challenge for developers and testers. Some technologies do not scale well, do not support

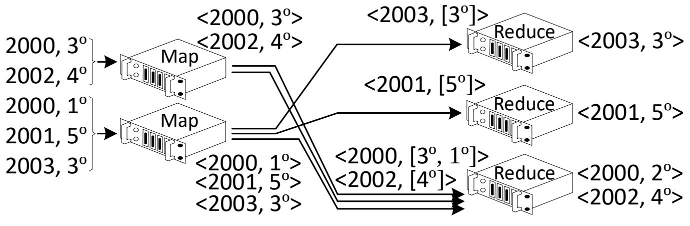

Fig. 3 Example of the MapReduce program that calculates the average temperature per year

indexing, or do not support ACID transactions, among others issues. Another challenge is the implementation of the data model in the program. *MapReduce* can analyze raw data without a data model (schema-less or unstructured) because the modelling of the data is codified in the program (schema-on-read). When considering the large data scale, it is difficult to establish a model for all data and there are several issues related to poor data quality, such as missing data, noise or incorrect data. Another problem is that new raw data are continuously generated and the data model could change over time, and then the program would need some changes.

The balance and the statistical properties of the data can also change over time and they can affect the program, especially if there are performance optimizations in the code based on data property assumptions. For example, suppose that in the program that analyzes the average temperature per year, the last two years contain 80% of the data. In this case there could be at least two issues: (1) performance problems if these two years are analyzed in the same computer, and (2) memory leaks or resource issues due to the high quantity of data analyzed by one computer. A further challenge is the type of processing implemented; originally *MapReduce* analyzed the data only in batches, but nowadays there are streaming or iterative approaches, among others. For example, the temperature sensors create streams of data, and so the calculation of the average temperature is more efficient using a streaming approach, but it is more difficult to implement and not all programs could be processed in this way. In some domains it is better to change the <key, value> approach to another that permits better modeling of the program, such as Pangool [56], that uses tuples, or more complex structures like graphs [57].

In the main framework of *MapReduce*, *Hadoop*, there are a lot of configuration parameters that could affect the execution in terms of resources, data replications and so on. More than 25 of these parameters are significant in terms of performance [58]. The developer does not know the resources available when the program is deployed because the cluster continuously changes (new resources adding to scale or infrastructure failures [59]), and this also makes the optimal configuration difficult. There are other advanced functionalities of *MapReduce* that could optimize the program, such as for example the *Combine* function. The problem is that if these functionalities are not well established there could be some side effects, such as incorrect output.

In *Big Data* there are also other testing issues related to the ethical use of data. Different security procedures and policies should be considered in *MapReduce* programs throughout the data lifecycle. For example, the analysis of some data could be forbidden in the next season due to agreements with the data provider or due to legal issues. In other cases, the data should be anonymized or encrypted, especially any sensitive data.

Several generic tools are used in the industry to test *MapReduce* programs, such as JUnit [60] with mocks. In order to facilitate the testing of *MapReduce* programs, MRUnit [61] runs the unit test cases without a cluster infrastructure. Another approach is MiniCluster [62] that simulates a cluster infrastructure in memory, or Herriot [63] that interacts with real infrastructure allowing finer-grained control, for example by the injection of computer failures that alter the execution of the program. There are different types of infrastructure failures that affect test execution and several tools simplify their injection such as AnarchyApe [64], ChaosMonkey [65] or Hadoop Injection Framework [66]. The remainder of this chapter analyses and summarizes the efforts of the research studies that are focused on covering the issues related to testing *MapReduce* applications.

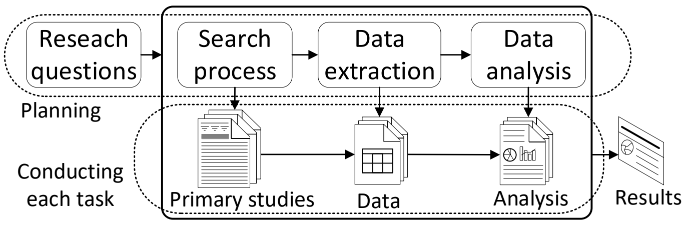

Fig. 4 Steps of systematic mapping study

II.2 Planning of the Mapping Study

This *mapping study* aims to characterize the knowledge of software testing approaches for *MapReduce* programs through a study of the exisiting research literature. To avoid bias, the planning of the *mapping study* describes several tasks based on the guidelines from Kitchenham et al. [67]:

1. Formulation of the research questions (Subsection II.2.1).
2. The search process to extract the significant literature (primary studies) to answer the research questions (Subsection II.2.2).
3. Data extraction to obtain the relevant data from the literature (Subsection II.2.3).
4. Data analysis to summarize, mix and put the data into context to answer the questions (Subsection II.2.4).

These tasks are planned and then conducted independently as described in Fig. 4. The execution (conducting) of the mapping study is summarized in Section II.3.

II.2.1 Research Questions

The research questions are formulated to cover all of the information about software testing research in the context of *MapReduce* programs with different points of view. This chapter formulates the research questions based on the 5W+1H model [68], [69], also known as the Kipling method [70]. This method is used in other *systematic reviews* of software engineering [71] and answers the questions: Why, What, How, By whom, Where and When.

The research questions of this *mapping study* are:

RQ1. **Why** is testing performed in *MapReduce* programs?

RQ2. **What** testing is performed in *MapReduce* programs?

RQ3. **How** is testing performed in *MapReduce* programs?

RQ4. **By whom**, **where** and **when** is testing performed in *MapReduce* programs?

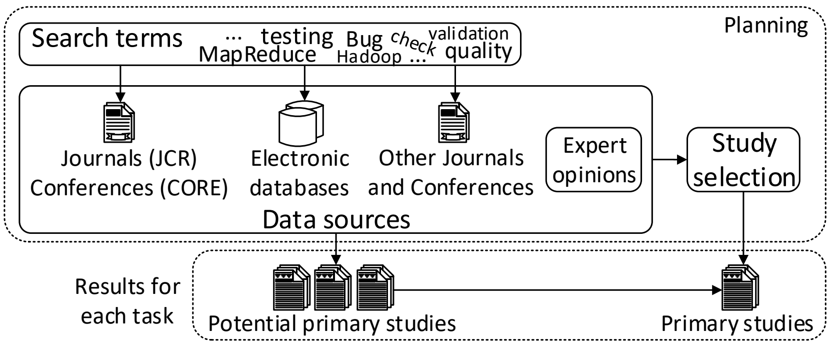

Fig. 5 Search process to obtain the primary studies in the mapping study

II.2.2 Search Process

The *mapping study* answers the research questions by analyzing the series of studies that contain relevant information about these questions. These studies are called primary studies and are obtained through the tasks described in Fig. 5. First, the search terms (set of several words/terms) related to software testing and *MapReduce* are searched for in different data

Table 1 MapReduce technology-related terms (population)

Technology	**Terms and years of creation**
Field	Big Data, Massive data, Large data
Data processing	Hadoop (2006)
- Batch	MapReduce (2004)
- Iterative	Spark (2013), Tez (2013), Stratosphere (2010), Dryad (2007), Flink (2014)
- Streaming	Storm (2011), S4 (2010), Samza (2013)
- Lambda	Lambdoop (2013), Summingbird (2013)
- BSP	Giraph (2013), Hama (2011)
- Interactive	Drill (2012), Impala (2012)
- MPI	Hamster (2011)
Testing	MRUnit (2009), Junit (1998), Mock, MiniMRCluster (2006), MiniYarnMRCluster (2012), Mini cluster (2007), QuerySurge (2011)
Security	Sentry (2013), Kerberos (2007), Knox (2013), Argus (2014)
Resource Manager	Yarn (2012), Corona (2012), Mesos (2009)
MapReduce abstraction	Pig (2008), Hive (2010), Jaql (2008), Pangool (2012), Cascading (2010), Crunch (2011), Mahout (2010), Data fu (2010)
Yarn frameworks	Twill (2013), Reef (2013), Spring (2013)
Yarn integration	Slider (2014), Hoya (2013)
Data integration	Flume (2010), Sqoop (2009), Scribe (2007), Chukwa (2009), Hiho (2010)
Workflow	Oozie (2010), Hamake (2010), Azkaban (2012), Luigi (2012)
Coordinator	Zookeeper (2008), Doozerd (2011), Serf (2013), Etcd (2013)
SDK	Hue (2010), HDInsight (2012), Hdt (2012)
Serialization	Sequence File (2006), Avro (2009), Thrift (2007), Protobuf (2008)
Cluster Management	Ambari (2011), StackIQ (2011), Whte elephant (2012), Ganglia (2007), Cloudera manager (2011), Hprof (2007), MRBench (2008), HiBench (2010), GridMix (2007), PUMA (2012), SWIM (2011)
Filesystem	HDFS (2006), S3 (2006), Kafka (2011), GFS (2003), GPFS (2006), CFS (2013)
Other storage	HBase (2008), Parquet (2013), Accumulo (2008), Hcatalog (2011)
Cluster deployment	Big top (2011), Buildoop (2014), Whirr (2010)
Data Lifecycle	Falcon (2013)

sources (journals, conferences and electronic databases). The papers that match these searches together with other studies recommended by experts constitute the potential primary studies. Finally, these studies are filtered as part of study selection in order to obtain only the studies that contain information which answers the research questions. In the following subsections each of the planning steps is described in detail.

II.2.2.1 Search Terms

The search terms are obtained from the three points of view proposed by Kitchenham et al. [67]: population, intervention and outcome. In this *mapping study* the population refers to the technologies and areas related to *MapReduce*, whereas the intervention and outcome refer to the software testing methods and the improvements obtained through software testing.

The search terms of this *mapping study* follow the chain "*MapReduce* technology related terms AND Quality related terms" where:

- **The *MapReduce* technology related terms** correspond with population and are enumerated in Table 1 with synonyms. The selection of the search terms is difficult when technologies are relatively new because the terminology is not well-established [33]. The *Big Data* paradigm and the *MapReduce* processing model are surrounded by a lot of buzzwords like other fields such as Cloud computing. In order to obtain the maximum

Table 2 Quality-related terms (outcome and intervention)

Quality characteristics	**Terms**
Functional suitability	Functionality, functional, suitability, suitable, correctness, correctable, accuracy, accurate, compliance, compliant, appropriateness, appropriate
Performance efficiency	Performance, performable, efficiency, efficient, time-behaviour, resource utilization
Compatibility	Compatibility, replaceability, replaceable, coexistence, interoperability, interoperable
Usability	Recognizability, recognizable, learnability, learnable, operability, operable, ease of use, helpfulness, helpful, attractiveness, attractive, attractivity, technical, accessibility, accessible
Reliability	Reliability, reliable, availability, available, fault tolerance, recoverability, recoverable
Security	Security, secure, safety, confidentiality, confidential, integrity, nonrepudiation, accountability, accountable, authenticity, authenticable
Maintainability	Maintainability, maintainable, modularity, modular, reusability, reusable, analyzability, analyzable, changeability, changeable, modification, modifiable, stability, stable, testability, testable
Portability	Portability, portable, adaptability, adaptable, transferability, transferable, installability, installable, effective, effectiveness
Other terms	Testing, assert, assertion, check, checking, test, test case, validate, validation, verify, verification, bug, defect, fault, failure, error, quality, risk, evaluation

relevant literature and avoid missing some primary studies due to buzzwords and jargon, a thorough search is performed considering the *MapReduce* and *Big Data* related technologies enumerated in Table 1.

- **Quality related terms** correspond with the Quality (sub)characteristics of ISO/IEC 25010:2008-2011 [72] and ISO/IEC 9126-1:2001 [73] and their synonyms (outcome), together with other testing terms (intervention). Both are enumerated in Table 2.

This chapter plans a wide search with 9384 combinations of terms in the paper title, obtained by 92 *MapReduce* technology-related terms and 102 quality-related terms.

II.2.2.2 Data Sources

The potential primary studies may be found in different data sources. This *mapping study* searches for the studies in the following data sources, grouped in four categories:

a) **High-impact journals and conferences**. The potential studies are obtained through DBLP [74] with the search terms in 31 JCR journals [75] and 53 CORE conferences [76] enumerated in Appendix VIII.1. The journals and conferences selected are related to the software testing or *Big Data*.

b) **Electronic databases**. The search terms are queried in IEEE Xplore [77], ACM Digital Library [78], Scopus [79], Ei Compendex [80] and ISI Web of Science [81], that are employed in other *mapping studies* of software testing [82] and recommended by Kitchenham et al. [83].

c) **Other journals and conferences**. Relatively new topics like *MapReduce* and *Big Data* are more likely to be published in specialized workshops/conferences [33]. The non-JCR journals and non-CORE conferences related to software testing or *Big Data* could be a good source of potential

II.2.2.3 Study Selection

Study selection is more difficult in *systematic mapping studies* than in *systematic reviews* [83]. Some potential primary studies obtained from the data sources might not contain information about software testing in the *MapReduce* programs. In this *mapping study* a series of filters selects only the studies that contain relevant information that answers the research questions. The potential primary studies that do not pass the filters are excluded, and the remainder make up the primary studies used to answer the research questions. The filters consist of the following exclusion criteria applied in the following order:

C1) Exclusion filter by year. A potential primary study is excluded when the publication year is before the *MapReduce* paper (2004) or before the creation of technologies/fields expressed in the search terms of Table 1.

C2) Exclusion filter by area. Potential primary studies are excluded when their research is not about Computer Science or Information systems.

C3) Exclusion filter by field. Potential primary studies are excluded when they do not contain *Big Data* information.

C4) Exclusion filter by topic. The final filter only includes the studies about software testing in the *MapReduce* programs; the remainder are excluded.

For example, the last filter excludes papers focused on software testing of the underlying technology such as the distributed system *Hadoop*, cloud computing, net or operative system, among others. Despite the normal execution of *MapReduce* programs depends on all these technologies, usually they are mature enough and the developer/tester is only focused on the *MapReduce* application. Some papers that have been excluded are intended to improve the performance of *Hadoop* through infrastructure failure forecasting [84] or to inject infrastructure failures in a distributed file system [85], among other examples that also do not test the *MapReduce* applications. Some other papers employ the *MapReduce* and *Big Data* capabilities to speed up testing in other *non-MapReduce* programs. For example, [86], [87] are frameworks to perform unit testing and mutation testing in general programs taking advantage of the parallel capabilities of the *MapReduce* processing model.

II.2.3 Data Extraction

The relevant information from the primary studies is extracted through a template divided in two parts. The first part is in general based on checklists of international standards related to the research questions, and the second part is focused on other data that could be interesting to analyze. The data extracted for answering the research questions are:

RQ1 "Why is testing performed in *MapReduce* programs?" Extraction of the arguments employed in the primary study to perform testing in *MapReduce* programs.

RQ2 "What testing is performed in *MapReduce* programs?" The data are extracted following two checklists that characterize the type of testing performed in each primary study: a checklist of the 31 ISO/IEC 25010:2011 Quality (sub)characteristics [72], and a checklist of the 17 ISO/IEC/IEEE 29119-4:2015 Quality-Related Types of Testing [88].

RQ3 "How is testing performed in *MapReduce* programs?" The data are extracted by following a checklist of the 11 ISO/IEC/IEEE 29119-1:2013 Annex A: Test activities [30], together with a checklist of test areas as follows: Testing specific to *MapReduce* programs, Testing not specific to *MapReduce* programs (other technologies/paradigms can be tested), Unclear and Not applicable. In addition, the following information about the tools used for testing is extracted: Does the study include the creation of a specific tool or use an existing tool? Is the tool based on another tool? Is the tool available? For example, if the tool is accessible via the Internet or with some type of open source license.

RQ4 "By Whom, where and when is testing performed in *MapReduce* programs?" The data are extracted following three checklists focused on the roles, the lifecycle and the test level. The first checklist contains the following roles: Manager, Analyst, Architect, Tester, Test manager, Test strategist, Other stakeholders, Unclear and Not applicable. These test roles are described

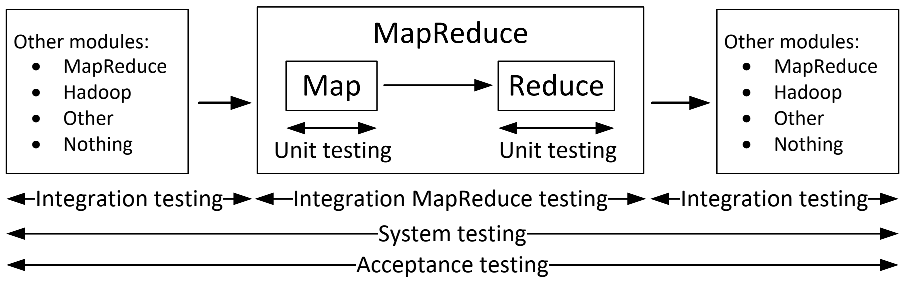

Fig. 6 Test levels based on ITSQB and adapted to MapReduce

in the ISO/IEC/IEEE 29119-1:2013 Annex E [30]. The second checklist contains the 6 ISO/IEC 12207:2008 Software Implementation lower level Processes [89] and the 11 ISO/IEC 12207:2008 System Context Technical processes [89]. The third checklist is based on ISTQB test levels [90] and adapted to *MapReduce* with two changes represented in Fig. 6: (1) Unit testing is divided into "Unit testing in *Map* function" and "Unit testing in *Reduce* function", and (2) "Integration testing" is for the integration of the *MapReduce* program with other modules, whereas "Integration *MapReduce* testing" is for the integration between *Map* and *Reduce* functions.

Other data are extracted in the *mapping study* because they may be interesting when characterizing the results and obtaining new findings. These data are extracted in a checklist with the following information about the research validation of the studies:

a) The different types of validation summarized by Mary Shaw [91]: Analysis, Evaluation, Experience, Example, Persuasion and Blatant assertion.
b) Other characterizations of the research: Validation with external programs, Validation with own programs, Another type of validation, Without validation, Unclear, Other and Not applicable.

II.2.4 Data Analysis

The data extracted from the primary studies are analyzed in order to answer the research questions. In empirical software engineering there are several methods [92] based on different approaches according to the type of data or research questions, among other things. In this *mapping study* the analysis is performed using (1) thematic analysis [93] to answer RQ1, and (2) meta-ethnography [94] for the remaining research questions. These methods are focused on qualitative data but analyze the data in a different way.

The thematic analysis method is selected to respond to RQ1 (Why is testing performed in *MapReduce* programs?) because it extracts a taxonomy of the reasons for testing from the primary studies. Then RQ1 is answered by a frequency analysis of these reasons for testing. This thematic analysis is performed with a grounded approach [95] that consists of the following steps:

1. Reading of the primary studies.
2. Extraction of the segments/phrases that include the reasons for testing.
3. Creating a group of labels for each previous segment/phrase based on the type of reason for testing.
4. Refining all labels several times until a few labels are obtained that compose a taxonomy of the reasons for testing.
5. Frequency analysis of the reasons for testing employed in the primary studies based on the previous taxonomy.

Meta-ethnography is selected to answer research questions RQ2 to RQ4 because it transforms the data from the primary studies into a more easily analyzable shared context. This method is employed in software engineering [96] and translates all primary studies on data under several facets that contain the checklists described in the data extraction (Subsection II.2.3). Once the data are extracted from the primary studies in these checklists, the research questions are answered by a frequency analysis. This *mapping study* follows the 7 steps proposed by Noblit et al. [94]:

1. Getting started. The topic under analysis is software testing of the *MapReduce* programs and is well studied through *mapping study*.
2. Deciding what is relevant to the initial area of interest. All primary studies are important.
3. Reading the studies. The primary studies are read in order to extract the relevant data.
4. Determining how the studies are related. Primary studies could contain related concepts or very different concepts. The relationship between these concepts is established through the checklists of the data extraction of Subsection II.2.3.
5. Translating the studies into one another. The primary studies are translated into relevant data according to the unified checklists of Subsection II.2.3.
6. Synthesizing translations. This *mapping study* creates more general concepts by the answers of research questions. RQ2 is answered by a frequency analysis of their two checklists, whereas both RQ3 and RQ4 are answered through their three checklists described in Subsection II.2.3.
7. Expressing the synthesis. The research questions are answered and discussed in Section II.4 following the previous steps of the *mapping study*.

II.3 CONDUCTING THE MAPPING STUDY

This section describes how each step of the *systematic mapping study* was conducted and how all problems were overcome. The planning of the systematic mapping study was refined by the three authors after several iterations.

Search terms: In the first instance, a small number of specific search terms such as *MapReduce* and *Big Data* were defined, but some relevant literature did not match with this search. For example, *Hadoop* is a distributed system that supports the execution of *MapReduce* programs and *non-MapReduce* programs, but there are several papers that use *Hadoop* and *MapReduce* words interchangeably. Other relevant papers do not include the word *MapReduce* in the title, but do contain other words related to the *MapReduce*/*Big Data* ecosystem like Hive, PIG or Spark. Finally, we refined the research method by adding more search terms in order to obtain the maximum amount of relevant literature.

Data sources: The data sources were also refined several times, especially the journals and conferences/workshops. Initially, we planned to analyze only the top journals and conferences such as ICSE. However, we observed that the relevant literature of software testing in *MapReduce* were not published at all in these journals and conferences. Finally, we added more journals and conferences/workshops that might contain relevant literature using both SEWORLD [97], DBLP [74] and our research experience. We added both JCR/CORE and non-JCR/non-CORE venues because a significant number of primary studies are published in this heterogeneity of venues, as we discuss in Section II.4.

Study selection: For each data source, one author developed queries using the large number of search terms. This search was difficult to carry out because the software engineering search engines did not adequately support the *mapping studies* searches [98]. To avoid this problem, we created a program that splits the 9384 combinations of search terms in 2346 searches and simulates a human performing these requests. The potential primary studies were obtained over a period of approximately two months in order to avoid bans in the search engines due to a high number of requests. After some months we tried to use this program in another *mapping study*, but the program was obsolete due internal changes in the search engines. As other researchers have noted, we also observe that digital search engines are not well-suited to complex searches [83].

After two months of both automatic and manual searches in 2311 proceedings/volumes (624 from JCR/CORE venues and 1687 from non-JCR/non-CORE venues), in July 2016 we obtained more than 100000 studies represented in Fig. 7. Then we removed those that were retrieved several times across different data sources, obtaining thereafter more than 70000 potential primary studies. The majority of these studies were clearly non-relevant for this *mapping study* because they were not focused on software testing in the *MapReduce* programs. Following some practices of other *systematic reviews* of both social science [99] and software engineering [100], those studies that were clearly non-relevant were filtered out by only one of the authors, whereas those studies that were potentially relevant were filtered in parallel by two of the authors. The first filter was applied by only one of the authors because it only excludes those studies that are either published before *MapReduce* or before the technology that matches the query. For example, there were several studies excluded in the first filter because despite the fact that they were retrieved by the words “testing” and “pig”, they were published before the Apache Pig technology (2008) was developed. These studies were usually concerned with testing pigs (the animals) rather than Pig (the software). The majority of studies could be excluded/selected after only reading the title, but in other cases the author needed to read the abstract or the whole paper, in particular when considering the last filters. After the first filter, there were still more than 14000 potential primary studies in consideration.

The second filter excludes those studies that are not related to either computer science or information systems. This filter was also applied by only one author because the studies excluded are clearly non-relevant, such as those about testing pigs (the animals) published after

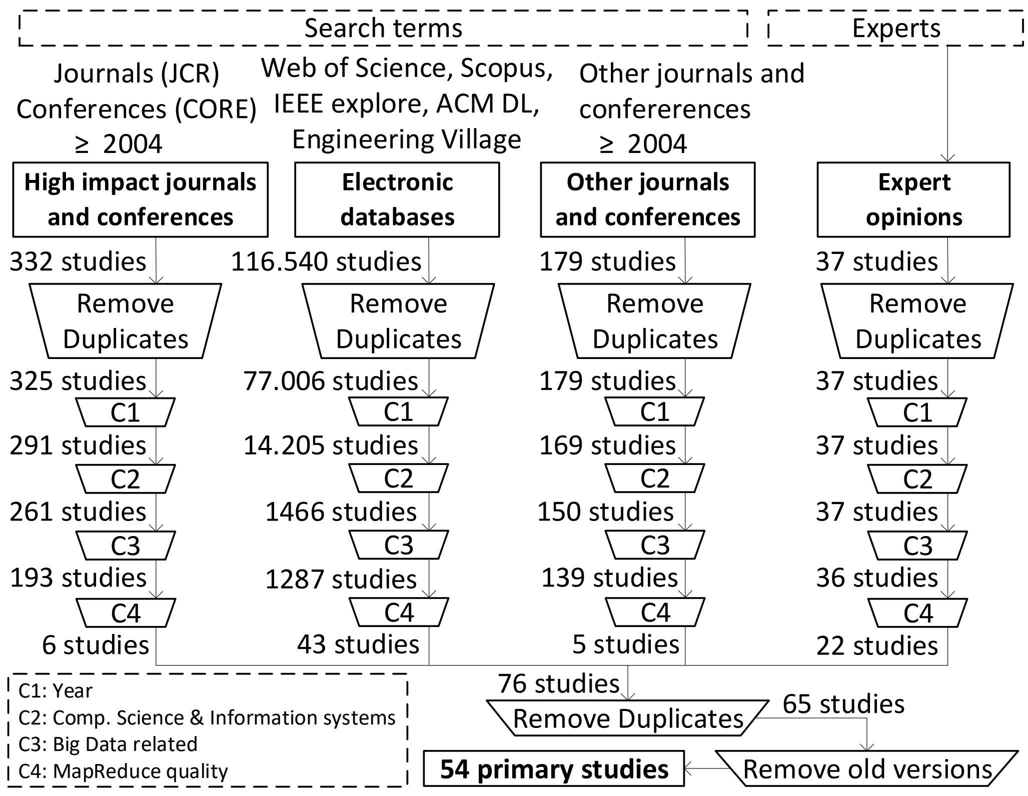

Fig. 7 Study selection of the primary studies

2008. After the second filter was applied, there were still more than 1500 potential studies. The third filter excludes those studies that are not related to the *Big Data* field. This filter was applied by one author and excluded a few studies, some of which are about "cascading failures" in computer science models or databases that are clearly unrelated to the *Big Data* field. After applying the third filter, there remained more than 1300 potential primary studies.

The fourth filter obtains those studies focused on software testing in the *MapReduce* processing model. This filter and the selection of the primary studies were almost completely applied by two of the authors and the disagreements were discussed by all authors. In the first instance one of the authors excluded 334 studies that are non-relevant because are related to *Big Data Analytics*. The remaining studies, numbering 1043, were related to *Big Data Engineering* and were filtered independently by two of the authors until the primary studies to be used in this chapter were obtained. Both authors agreed on 1002 studies: 50 of them passed the filter and were selected as primary studies, and the other 952 did not pass the filter. In contrast, both authors disagreed on 41 studies: one of the authors considered that 35 of them should pass the filter and be selected as primary studies, whereas the other author considered that the other 6 studies should also pass the filter and be selected as primary studies. Despite 96% agreement between both authors, we applied the Kappa coefficient to statistically measure the inter-rater agreement [101]. We obtained 0.69 as a Kappa coefficient with [0.60-0.78] as a 95% confidence interval. This is usually interpreted as substantial [101] or moderate [102] agreement between both authors during the selection of the primary studies. The 4% disagreement (representing 41 studies) were discussed and analyzed by the three authors until total agreement of the primary studies to be used in this chapter was achieved. The majority of disagreements were caused by an initial incorrect definition of the *systematic mapping study* plan because one author considered that studies about software testing in Hadoop system should be considered as primary studies, and the other author did not. We refined the plan indicating that the primary studies are only those about software testing in the *MapReduce* processing model and not those about software testing in other technologies or frameworks that do not comprise *MapReduce*. Other disagreements happened because one of the authors did not consider those papers about software testing in *MapReduce* abstractions like Pig and Hive as primary studies. There were other disagreements, for example those papers that instead of testing are related to debugging. After all authors had discussed and resolved the disagreements, 65 studies passed the filter. Some of these studies are the continuation of the same research, such as a conference paper with an improvement published in a journal. The old versions of the studies were excluded keeping only the latest study. There were several papers from the HP Labs team, but we considered that only three of them are considered primary studies because these studies were distinct from each other. As Section II.4 discussed, one of them [103] is focused on obtaining the execution time with microbenchmarks, whereas the other [104] is focused on the cloud cluster using different techniques, and the final study [105] is focused on Pig queries. Finally, 54 unique studies were selected as primary studies.

<u>Data extraction</u>: In order to perform the data extraction, each one of these 54 primary studies were read at least once by two of the authors. Despite the guidelines from Kitchenham et al. [67] which suggest that at least two researchers extract the data independently, other researchers consider it practical that one author extracts the data and the other author checks the extraction [98]. This last practice is applied in software engineering by other *systematic reviews* [106] and we also extracted the data in similar way. One of the authors extracted the data from the primary studies, another author checked the extraction, and the doubts were discussed and resolved by the three authors.

Data analysis: Once the data were extracted, all authors discussed the interpretations and potential results. Then the three authors started to write the findings and the report.

The current *systematic mapping study* took a lot of time despite not being the first conducted by our research group. The time consumed is one of the main criticisms of *systematic reviews* [83] .We specifically expended more time in: (1) creation and execution of a program to support the high number of search queries, (2) selection of the primary studies from a large amount of literature, (3) extraction of the data from each primary study, and (4) refinement of the research method. We performed the *systematic mapping study* two times, initially in 2015 and finally updated with the literature of 2016.

II.4 RESULTS

The results were obtained through the execution (conducting) of the *systematic mapping study* that answers the research questions. The primary studies are summarized in Subsection II.4.1. From these, the data were extracted, and the analysis is developed in Subsection II.4.2 answering the research questions. Other results that do not answer the research questions but remain relevant in characterizing the state-of-art of software testing in *MapReduce* applications are summarized in Subsection II.4.3. Finally, the general results are discussed in Subsection II.5.

II.4.1 Primary Studies

In this chapter, there are 54 primary studies that are derived from more than 70000 potential studies obtained though the search process detailed in Fig. 7. These primary studies are detailed in Appendix VIII.3 with the year of publication, type of contribution and a summary of their contents.

The *MapReduce* processing model was described in 2004, but the software testing efforts in this field according to the primary studies only started in 2010 with only 1 study and after six years and six months the number of primary studies had increased to 54. Table 3 summarizes the frequencies of these primary studies over time and reveals that the research efforts of the topic may have grown because after 2013 the attention increases.

The different types of validations employed in the research are summarized in Table 4. The majority of the studies validate their research through examples (41%) or experience (35%). In 76% of the studies, the validation is carried out by applying the testing research in a program(s), but in 11% of the primary studies the research is not validated.

Testing in *Big Data* has opened up new challenges [107], especially in the understanding of the data and its complex structures [108]. Gudipati et al. [109] establish a classification of testing in the *Big Data* field. This study includes the validation of the *MapReduce* process together with other non-functional characteristics like performance and failover. All of these characteristics are among the main challenges in *Big Data* testing [108] . In order to overcome these challenges

Table 3 Frequency of the primary studies over time

Statistics	2010	2011	2012	2013	2014	2015	2016 until July
Frequency	1 (2%)	7 (13%)	4 (7%)	19 (35%)	12 (22%)	10 (19%)	1 (2%)
Absolute frequency	1 (2%)	8 (15%)	12 (22%)	31 (57%)	43 (80%)	53 (98%)	54 (100%)

Table 4 Number of primary studies per type of validation

			Number of studies			
Analysis					7 (13%)	54 (100%)
Evaluation					0 (0%)	
Experience					19 (35%)	
Example					22 (41%)	
Persuasion					0 (0%)	
Blatant assertion					6 (11%)	
With validation	**Over programs**	**External programs**	30 (56%)	41 (76%)	47 (87%)	54 (100%)
		Own programs	12 (22%)			
	Other validation			8 (15%)		
Without validation				6 (11 %)		

though software testing, it is recommended to deploy a distributed environment like production, preferably in the cloud [109], [110].

Software testing can be performed in different dimensions and some authors suggest addressing the three Vs of *Big Data* (Volume, Velocity and Variety). In the case of high Volume, it could be difficult to check whether the test case output is the output expected, and the use of automatic tools can be helpful [109]. In the case of Variety such as semi-structured or un-structured data, it can be helpful to transform them in a structured way [109]. To test the Velocity, it is recommended to design performance tests [109]. In addition to Volume, Variety and Velocity, other authors suggest considering the Veracity through data cleaning and normalization [110]. Those four Vs have an impact not only on the program execution, but also on the performance tests [111]. Zhenyu Liu [111] classifies the performance testing in *Big Data* as: (1) concurrent test (the impact of multiple users and applications in concurrency), (2) load testing (realistic data loads to analyze the response of the program), (3) stress test (testing under extreme data), and (4) capacity test (the analysis of the resources that can be used).

The majority of the primary study papers are focused on capacity and load testing. These studies are summarized in Subsection II.4.1.1, whereas those primary studies that are more related to the functionality are described in Subsection II.4.1.2.

II.4.1.1 Performance testing and analysis

In the primary studies, performance analysis is mainly addressed by the simulation of program executions, or by evaluation of a performance prediction model. These prediction models characterize performance based on different kinds of input parameters. The model of Song et al. [112] predicts the execution time given some characteristics about both the input dataset, the program functionality and the programming cluster. In addition, other models obtain the execution time by also considering the file system [113]. The prediction models can have different goals beyond the execution time, for example the Yang et al. model [114] helps to obtain the values of the input parameters that achieve the best execution time. The tester varies the input parameters (the network or the locality of the data, among others) and then analyzes the impact in performance.

Performance can be predicted by using a stochastic approach, for example by Stochastic PetriNets [115]. Another stochastic model [116] also considers the *MapReduce* tasks that are re-executed due to frequent failures. The performance of *MapReduce* and *Big Data* applications can also be evaluated through large scale stochastic models by Mean Field Analysis [117].

While some models predict performance by analyzing the execution time of several samples [118] or considering previous executions [103], other models consider some specific characteristics of the *MapReduce* execution. The Vianna et al. model [119] considers the influence over the performance of *MapReduce* tasks that are executed in parallel. The network is another issue that can cause bottlenecks in *MapReduce* programs and several models consider the network in order to predict the performance [120], [121]. Others also consider the task failures and I/O congestion [122].

Together with the network, memory can cause performance issues, especially in iterative programs or those with high I/O operations. The performance of the shared-memory computation programs can be predicted with the Tanzil et al. model [123], whereas in those programs with Remote Direct Memory Access, the Wasi-ur-Rahman et al. model [124] can be used. Apache Spark 3 programs process the data using distributed memory abstraction and their performance can be predicted by a model that executes a sample of data [125].

The cluster that executes *MapReduce* programs can also influence performance, especially when this cluster is formed by a heterogenic infrastructure. In these clusters, the Zhang et al. model [104] predicts performance within the bounds of upper and lower execution time. Another model that can predict the performance in these clusters employs the machine learning technique Support Vector Machine [126]. There are several clusters deployed in the cloud to obtain several advantages in terms of elasticity and cost. For programs executed in these clusters, performance can be predicted modeling the systems with Layered Queueing Network [127]. In the case of I/O intensive programs in the cloud, performance can be predicted using a CART (Classification And Regression Tree) model [128]. When the programs executed in the public cloud have deadline requirements to satisfy, performance can be predicted with the Locally Weighted Linear Regression model considering the previous execution and the data executed in parallel [129]. For those programs that are not only executed in a public cloud, but in a hybrid cloud, their performance can be predicted with the Ohnaga et al. model [130].

Several frameworks transform queries into *MapReduce* jobs, such as Hive [131] and Pig [132]. The execution time of the Hive SQL-like queries may be forecast using multiple linear regression to predict the execution time of all the *MapReduce* jobs generated from these queries [133]. The multiple regression analysis can be also used to predict the execution time of the join queries in Pig programs [134]. In contrast, the Zhang et al. model [105] predicts the performance of Pig programs considering the previous executions.

In addition to the prediction models, the testers can simulate the execution of the programs to analyze their performance in a fine-grained way. As with the prediction models, the simulators also consider characteristics about the input dataset, the program functionality, the programming cluster and the file system [135]. The MRPerf simulator [136] considers the inter and intra rack interactions over network using ns-2, and can be combined with other simulators, such as DiskSim. The Chauhan et al. simulator [137] is based on MRPerf but including, among others elements, some random time due to operating system scheduling and network communication delays.

The execution time of *MapReduce* programs can also be obtained using the modelling language proposed by Barbierato et al [138]. The tester can also monitor the execution of *MapReduce* programs and test cases, obtaining charts to evaluate performance and potential bottlenecks [139]. Villalpando et al. [140] propose a model for the *Big Data* application establishing a relationship between performance and reliability measures based on the international standard of quality ISO/IEC 25010 [72].

Despite there being several research lines concerned with predicting the execution time, there is no comprehensible comparison between them. In general, these studies are evaluated only with a few different case studies. The scientific contribution of these prediction models can be improved with empirical evaluation against other models using a standardized benchmark.

The main difference between these models is not just the technique/approach employed, but also the parameters used by the model. Different characteristics of the input dataset, program functionality, programming cluster and file system are considered as parameters, for example: size of data or number of <key, value> pairs (input dataset), complexity or overhead of Map (program functionality), number of CPU cores or racks (programming cluster), and number of HDFS replicas or the data transfer time for an HDFS block (file system).

There are a lot of different parameters, but there is no clear indication of which parameters have more influence on performance. The contribution of the performance prediction studies can be improved evaluating which parameters really influence performance and which do not. Then the prediction models can be designed with a more standardized subset of parameters that have a notorious influence on performance.

II.4.1.2 Functional testing

Misconfiguration is one of the most common problems that lead to memory/performance issues in *MapReduce* [141]. However, according to the empirical study by Ren et al. [28], users rarely tune the configuration parameters that are related to performance. Users usually only turn the configuration parameters that are related to failures [28]. Another empirical study analyzes 200 production failures and determines that the majority of failures are related to the data, and only 1.5% are related to the performance (out of memory) [142]. In production there are several programs that do not finish their execution; Kavulya et al. [27] indicate that around 3% of programs have this problem, and a broader study indicates this percentage falls between 1.38% and 33.11% [28].

An analysis of 507 programs indicates at least 5 different kinds of faults caused by the non-deterministic execution of the *MapReduce* [29]. The current thesis is focused in these faults. Camargo et al. [143] classify the specific faults of *MapReduce*, whereas our initial work [38] classify those caused by the non-determinism execution. Chen et al. [144] propose a formal approach to detect these faults caused by non-determinism. In contrast, Csallner et al. [145] employs symbolic execution to check the program under test. Another technique to detect the faults caused by non-determinism dynamically checks the properties of the program under test with random data [146]. One of the reasons for the non-deterministic execution is the tolerance of infrastructure failures. There are several studies that propose to inject infrastructure failures in the test case design [147]. Failure Scenario as a Service (FSaaS) [148] injects infrastructure failures into a cluster deployed in the cloud.

Several testing techniques are devised in order to generate test inputs aimed at detecting functional faults, such as those caused by nondeterministic execution or other semantic errors. The MRFlow testing technique [53] generates the test coverage items that can be used to generate test inputs based on the data-flow technique adapted to the *MapReduce* processing model. Another technique to generate data of the test cases employs a bacteriological algorithm aimed to kill some semantic mutants specific to *MapReduce* which varies both the number of the *Reducers* and the existence (or not) of the *Combiner* functionality [149]. In those *Big Data* ETL (Extract, Transform and Load) programs that integrate several technologies (*MapReduce*, Pig, Hive, among others), a subset of representative data for test can be obtained from the dataset through input space partition together with constraints [150]. In dataflow programs like Pig, the test inputs can be generated using dynamic-symbolic execution in the control-flow graph of the program [151].

Other kinds of checks can be performed in *MapReduce* programs. Dörre et al. [152] propose an automatic checker that statically detects incompatibilities between the types of the <key, value> pairs processed by *MapReduce* programs. Rabkin et al. [153] statically analyze the configuration parameters used by different frameworks, including Hadoop. The *MapReduce* developers and testers should analyze the configuration parameters used because 17% of Hadoop options are not documented and 6% are not used in the code. The main *Big Data* frameworks can be affected in the same way as Hadoop because these issues are common in open-source programs [153]. The correctness of *MapReduce* programs can also be verified formally through proofs modelling the specification as Coq functions [154].

II.4.2 Analysis

The primary studies contain the answers to the research questions, but this information is hidden inside them. The analysis obtains valuable information in order to answer the research questions based on the data extracted from the primary studies. The data were extracted following the template defined in Subsection II.2.3 and then analyzed by the methods described in Subsection II.2.4. In the following subsections the primary studies are analyzed, classified and summarized in order to obtain the answer to each research question systematically.

II.4.2.1 RQ1 Why is testing performed in MapReduce programs?

MapReduce programs are tested for several reasons. A model/taxonomy of these reasons were obtained by applying the thematic analysis method to the primary studies, as described in Subsection II.2.4. The reasons for testing obtained are:

- *Performance related*: issues derived from the performance goals, service level agreements, size of the data, performance under infrastructure failures and prediction/analysis/optimization of performance.
- *Failure related*: the specific faults of *MapReduce* programs and the number of programs that fail in production.
- *Improper use*: not all programs fit correctly in the *MapReduce* processing model.
- *Data related*: the challenges related to schema-less data and poor data quality.
- *Configuration related*: the misconfiguration of the infrastructure or program parameters may produce a failure.
- *Time related*: the programs may fail after a long time of resource usage.
- *Cost related*: testing can be carried out in order to reduce the cost of development, resource utilization and so on.
- *Other*: the reasons that do not fall in another category of the model/taxonomy of the reasons but do not constitute a new category of reasons.

For each of the above categories of reasons for testing, Table 5 indicates the number of primary studies that details these reasons. Note that a primary study can contain one or more reasons

Table 5 Number of primary studies per type of reason for testing

Types of reasons	Number of papers	Number of formal reasons		Number of informal reasons		Number of total reasons	
Performance related	30	5	(6%)	36	(43%)	41	(49%)
Failure related	11	3	(4%)	9	(11%)	12	(14%)
Improper use	2	3	(4%)	1	(1%)	4	(5%)
Data related	9	2	(2%)	8	(10%)	10	(12%)
Configuration related	3	2	(2%)	1	(1%)	3	(4%)
Time related	2	2	(2%)	0	(0%)	2	(2%)
Cost related	7	0	(0%)	7	(8%)	7	(8%)
Other	4	0	(0%)	4	(5%)	4	(5%)
		17	(20%)	66	(80%)	83	(100%)

for testing. In Table 5, each reason for testing is also classified based on the degree of formality of the evidence in accordance with the following types: reasons with formal evidence and reasons with informal evidence.

Reason with formal evidence: the reason for testing is detailed in the primary studies empirically or with some rigorous evidence of this reason to test. For example, if one paper performs an extensive analysis of several programs and detects that testing is necessary because a lot of programs crash in production, this would be considered a reason with formal evidence.

Reason with informal evidence: the reason for testing is not clearly explained or not detailed in the primary studies due to the absence of rigorous analysis of the evidence for this reason to test. For example, if a paper indicates that the testing is necessary because the developers do not know how to configure the performance parameters of *MapReduce* programs, this would be considered a reason with informal evidence.

The most frequent type of reason for testing is "performance related", being described in 30 primary studies and representing 49% of the total reasons given in all primary studies, followed by "failure related" with 14%, "data related" with 12%, and "cost related" with 8% of the total number of reasons. Considering the formal evidence of the testing reasons, "performance related" is also the main reason in the primary studies with 6% of the total reasons (5 of formal evidence out of a total of 17 of formal evidence), followed by "failure related" and "improper use" with 4% of total reasons (3 of formal evidence out of a total of 17 of formal evidence).

In the model/taxonomy obtained through the analysis of the primary studies, the 41 "performance related" reasons for testing were sub-divided in the following sub-categories of reasons:

- *Optimization/improvement of application performance*: testing is aimed at the improvement of program performance.
- *Analysis of application performance*: understanding of performance to detect bottlenecks, among other issues.
- *Influence of the infrastructure on application performance*: whereas *MapReduce* applications can be designed without considering the infrastructure, program performance is influenced by the production infrastructure.

- *Influence of dataset on application performance*: in the same way that the infrastructure impacts performance, the dataset used in production also make an influence.
- *Fulfill SLA or performance goals*: the reason for testing the program is to fulfill service level agreements or other performance goals such as deadlines.
- *Other*: the reasons that do not fall in another sub-category of the model/taxonomy of the performance reasons but do not constitute a new sub-category of reasons.

For each one of the above sub-categories of testing that are related to performance, Table 6 indicates the number of the primary studies and their reasons for testing.

From the 41 "performance related" reasons for testing, the most frequent are focused on the analysis (27% of "performance related" reasons) and optimization of performance (27% of "performance related" reasons), followed by the fulfillment of performance goals (24% of

Table 6 Number of primary studies per type of performance-related reason

Types of "performance related" reasons	Number of papers	Number of formal reasons	Number of informal reasons	Number of total reasons
Optimization/improvement of application performance	11	0 (0%)	11 (27%)	11 (27%)
Analysis of application performance	11	0 (0%)	11 (27%)	11 (27%)
Influence of infrastructure in application performance	4	3 (7%)	2 (5%)	5 (12%)
Influence of dataset in application performance	2	0 (0%)	2 (5%)	2 (5%)
Fulfill SLA or performance goals	10	2 (5%)	8 (20%)	10 (24%)
Other	2	0 (0%)	2 (5%)	2 (5%)
		5 (12%)	36 (88%)	41 (100%)

Table 7 Number of primary studies per ISO/IEC 25010:2011 Quality (sub)characteristic

			Number of studies		
ISO 25010:2011 System/software product quality	**Functional suitability**	**Functional Completeness**	2 (4%)	14 (26%)	46 (85%)
		Functional correctness	14 (26%)		
		Functional appropriateness	2 (4%)		
	Performance efficiency	**Time-behaviour**	32 (59%)	35 (65%)	
		Resource utilisation	14 (26%)		
		Capacity	1 (2%)		
	Reliability	**Maturity**	1 (2%)	3 (6%)	
		Availability	1 (2%)		
		Fault tolerance	1 (2%)		
		Recoverability	3 (6%)		
Other studies	**Characterization studies**			4 (7%)	8 (15%)
	Overview of testing			4 (7%)	

"performance related" reasons). The remainder of reasons for testing related to performance analyze the influence of the infrastructure (12% of "performance related" reasons) and the dataset (5% of "performance related" reasons), followed by other issues (5% of "performance related" reasons).

Of all the reasons for testing the programs, only 20% are based on formal evidence, and the remaining 80% are based on informal evidence. Regardless of the formality of evidence, the reasons for testing *MapReduce* programs most frequently described in the primary studies include "performance related", especially for the analysis, optimization and fulfillment of performance goals. The least commonly cited reasons for testing are "time related", "configuration related", "improper use" and "other".

II.4.2.2 RQ2 What testing is performed in MapReduce programs?

The planning of Subsection II.2.4 proposes a meta-ethnography [94] to answer this research question. The data extracted from each primary study is categorized against two facets in order to answer RQ2:

a) Quality (sub)characteristics for each study according to the ISO/IEC 25010:2011 [72] represented in Table 7.
b) Quality-Related Types of Testing proposed in each study based on ISO/IEC/IEEE 29119-4:2015 [88] and summarized in Table 8.

The majority of efforts are focused on "performance efficiency", accounting for 65% of the studies, then on "functional suitability" with 26% of the studies, and finally on "reliability" with 6% of the studies. Regarding the type of testing, 59% apply "performance-related testing", 22% employ "functional testing" and 4% use "backup/recovery testing".

The results obtained through the combination of both facets are more or less those expected: "performance-related testing" is related to "performance efficiency" characteristics, the "functional testing" to "functional suitability", and "backup/recovery testing" to "reliability".

II.4.2.3 RQ3 How is testing performed in MapReduce programs?

This research question is answered through the meta-ethnography [94] proposed in Subsection II.2.4. In order to answer RQ3, the primary studies were analyzed considering three facets:

a) Testing methods/techniques are summarized in Table 9 according to the test activities proposed in Annex A of ISO/IEC/IEEE 29119-1:2013 [30].
b) Dependency between the primary studies and the *MapReduce* processing model is depicted in Table 10. This table describes whether the testing methods, techniques or studies are specific to *MapReduce* or could be applied to other paradigms/technologies.

Table 8 Number of primary studies per ISO/IEC/IEEE 29119-4:2015 Quality-Related Type of Testing

		Number of studies		
ISO/IEC/IEEE 29119-4:2015 Types of testing	**Performance-Related Testing**	32	(59%)	44 (81%)
	Functional Testing	12	(22%)	
	Backup/Recovery Testing	2	(4%)	
Other studies	**Characterization studies**	5	(9%)	10 (19%)
	Overview of testing	5	(9%)	

c) Tools created or used in the primary studies to perform software testing are characterized in Table 11.

The majority of the papers (74%) focus on testing only the *MapReduce*-specific parts of the program. These programs have challenges related to performance issues and the correct operation of the program under parallel architecture. These issues among others are tested mainly by “evaluation”, according to 48% of the studies and “simulation” in 17% of the studies. Other testing activities are used to a lesser degree, such as “structure based” in 7% of the studies or static analysis in 6% of the studies.

More than half of the studies (65%) do not create or use testing tools in their research. There are in total 19 tools, where 11 are based on other software testing related tools, and only 5 are freely available on the Internet with an open source license.

Table 9 Number of primary studies per ISO/IEC/IEEE 29119-1:2013 Test activity of Annex A

				Number of studies			
ISO/IEC/ IEEE 29119-1:2013 Annex A: Test activities	**V&V analysis**		**Evaluation**	26 (48%)		29 (54%)	43 (80%)
			Simulation	9 (17%)			
	Testing	**Dynamic testing**	**Structure based**	4 (7%)	7 (13 %)	12 (22%)	
			Specification based	1 (2%)			
			Experienced based	1 (2%)			
			Other	1 (2%)			
		Static testing	**Static analysis**	3 (6%)	5 (9%)		
			Other	3 (6%)			
	Formal methods		**Model checking**	1 (2%)		2 (4%)	
			Proof of correctness	1 (2%)			
Other studies	**Characterization studies**				6 (11%)	11 (20%)	
	Overview of testing				5 (9%)		

Table 10 Number of primary studies per test area covered

		Number of studies	
Specific of MapReduce		40 (74%)	50 (93%)
Not specific of MapReduce		10 (19%)	
Other studies	**Characterization studies**		4 (7 %)

Table 11 Number of primary studies per tool created in their research

			Number of studies		
Tool created or used	**Based on other**	**Tool available**	3 (6%)	11 (20%)	19 (35%)
		Tool not available	8 (15%)		
	Not based on other tools	**Tool available**	2 (4%)	8 (15%)	
		Tool not available	6 (11%)		
No tool created or used					35 (65%)

II.4.2.4 RQ4 By whom, where and when is testing performed in MapReduce programs?

The planning of the *mapping study* described in Subsection II.2.4 proposes a meta-ethnography [94] to answer the research question through three facets:

a) The different roles that participate in the testing efforts of the *MapReduce* programs, described in Table 12.
b) Test levels summarized in Table 13 that contains a characterization of ISTQB test levels [90] adapted to the *MapReduce* processing model according to Fig. 6.
c) The development cycle phase according to the Software Implementation lower level Processes and System Context Technical Processes of ISO/IEC 12207 [89] described in Table 14.

Table 12 Number of primary studies per role

		Number of studies	
Roles	**Tester**	45 (83%)	49 (91%)
	Developer	5 (9%)	
Other studies	**Characterization studies**		5 (9%)

Table 13 Number of primary studies per ISTQB test level

			Number of studies		
Levels of testing in ISTQB	**Unit testing**	**Unit testing Map**	16 (30%)	19 (35%)	44 (81%)
		Unit testing Reduce	19 (35%)		
	Integration MapReduce testing		35 (65%)		
	Integration testing		4 (7%)		
	System testing		2 (4%)		
	Acceptance testing		0 (0%)		
Other studies	**Characterization studies**			5 (9%)	10 (19%)
	Overview of testing			5 (9%)	

Table 14 Number of primary studies per ISO/IEC 12207:2008 Software Implementation Lower Level Process and System Context Technical Process

		Number of studies	
ISO/IEC 12207:2008 Software Implementation lower level Processes	**Software Construction Process**	3 (6%)	48 (89%)
	Software Qualification Testing Process	47 (87%)	
ISO/IEC 12207:2008 System Context Technical processes	**Implementation Process**	3 (6%)	48 (89%)
	System Qualification Testing Process	47 (87%)	
	Software Operation Process	1 (2%)	
Other studies	**Characterization studies**	5 (9%)	6 (2%)
	Overview of testing	1 2%)	

As expected, the main player for testing the *MapReduce* programs is the tester, as per 83% of the studies, and then the developer according to 9% of the studies. Almost all primary studies (87%) describe testing efforts in the “Software/System Qualification Testing Process” compared with 6% which focus on “Software Construction or the Implementation Process”. In these processes, the studies cover in more detail the specific *MapReduce* parts of the program (*Map* and *Reduce* functions) instead of the other parts. The majority of the research efforts in 65% of the studies focus on the integration testing between *Map* and *Reduce* functions, and then 35% of the studies cover unit testing at the *Map* or *Reduce* functions. To a lesser extent, the testing efforts are oriented towards the parts of the program that could not contain *MapReduce* functions: 7% of the studies consider integration testing between the *MapReduce* functions with other parts of the program, and 4% of the studies relate to testing the system. All testing levels are covered by the primary studies except for acceptance testing.

From these results, it appears that the fulfillment of the contract or user requirements tested in the acceptance testing level is not greatly affected by the existence of *MapReduce* functions in the system. Despite the fact that *Big Data* programs can contain a composite of several technologies/programs, testing research efforts focus on testing the *MapReduce* functions in isolation from the rest of the system. Few studies consider that a *Big Data* program can contain *MapReduce* functions together with other technologies. Regardless of the test level, the testing described in the primary studies is mainly performed in the Software/System Qualification Testing Process.

II.4.3 Summary

The research questions of Subsection II.2.1 were answered through the primary studies, data extraction and data analysis. A summary is presented below:

RQ1. Why is testing performed in *MapReduce* programs? There are at least seven reasons for testing the *MapReduce* programs. The most frequent reasons are based on performance issues (to analyze, optimize and fulfill performance goals), the existence of several or specific failures, the type and quality of the data processed by these programs, and testing to predict the resources required and efficiently select the resources to be used. To a lesser degree, the other reasons for testing are the improper use of the processing model or technology, program misconfiguration or failures after a long period of executions.

RQ2. What testing is performed in *MapReduce* programs? The majority of the research efforts in testing the *MapReduce* programs focus on the analysis of performance, and to a lesser extent the functional aspects of *MapReduce* programs.

RQ3. How is testing performed in *MapReduce* programs? Mainly by evaluation and simulation. In both cases testing is focused specifically on the *MapReduce* functions and does not consider other parts of the program. Several tools are used to perform testing, but few are available on the Internet.

RQ4. By whom, where and when is testing performed in *MapReduce* programs? Testing is mainly performed by the tester in the Software/System Qualification Testing Process and major efforts focus on the *MapReduce* program (unit and integration testing between *Map* and *Reduce* functions).

The analysis of several features about the primary studies reveals, in addition to the answers to the research questions, other findings which are analyzed below.

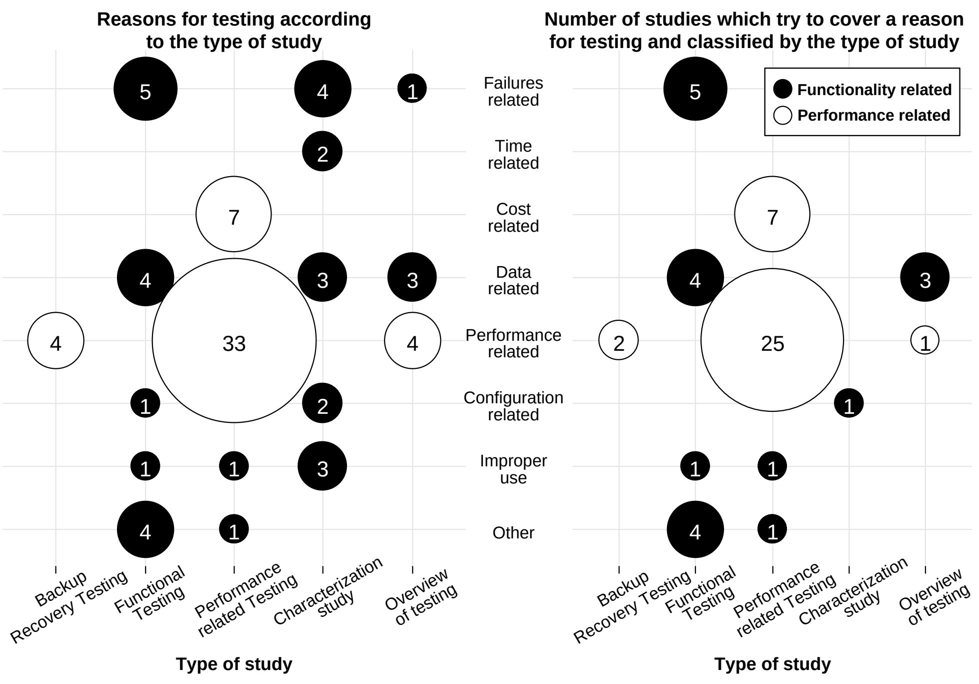

Fig. 8 Number of reasons for testing and primary studies per type of study

The relation between the reasons for testing the programs and the type of testing employed in each study is displayed in Fig. 8. According to Table 8, 59% of the studies focus on performance testing (RQ2), which is very important because *MapReduce* applications analyze large quantities of data. From RQ1 the reasons for testing the programs are obtained and 58% of these reasons are related to performance (48 reasons of a total of 85 according to the left side of Fig. 8). The reasons for performance testing and the number of studies that test performance are aligned. However, according to Table 8, the studies related to functionality only represent 22% of all studies even though 42% of the reasons for testing are related to functionality (35 reasons of a total of 85 according to the left side of Fig. 8). There are more reasons for testing functionality than there are studies about functionality, which can indicate a challenge in the functionality testing to cover these reasons and improve the quality evaluation of the *MapReduce* applications. The current thesis is focused on these functional faults through testing (Chapter III), debugging (Chapter IV) and operations (Chapter V).

The main test activities in RQ3 are evaluation (seen in 48% of the studies) and simulation (seen in 17%). These two activities are the most frequent because the majority of studies are focused on performance testing (59% according to RQ2). Fig. 9 characterizes the test activities (RQ3) and test levels (RQ4) regarding different types of testing (RQ2). The test levels in each type of testing are more or less similar to the answer to RQ4: the principal efforts are at the integration testing level of *Map* and *Reduce* functions and to a lesser degree at unit level. However, the test activities are different depending on the type of testing: performance testing employs evaluation and simulation to predict the time of execution and resources required, but functionality testing performs a variety of different test activities considering specific characteristics of the *MapReduce* processing model (static testing, structure based, formal methods, experience based and specification based).

The majority of the studies are published in conferences (76%) and there are a few studies published in a high-impact journal (13%). Despite the fact that the number of research lines of

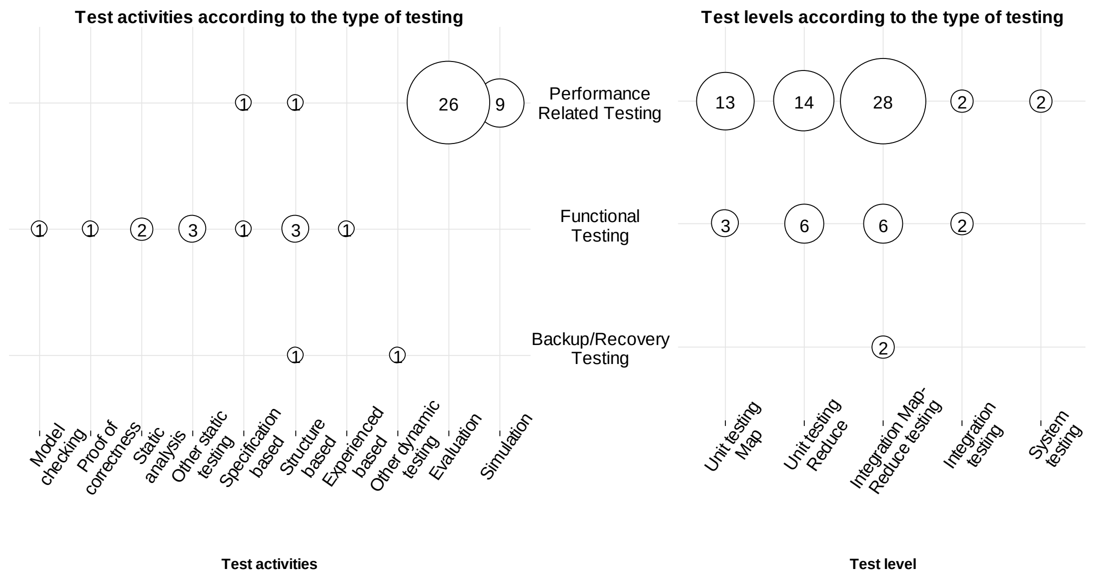

Fig. 9 Number of primary studies per test activity and test level according to the type of testing

testing in *MapReduce* is growing, the validation of these approaches is still simply through experience or case studies focusing on only a few programs, which are sometimes created by the researcher. According to Table 4, 11% of studies are not validated, 41% are validated with examples and 22% employ programs created by the researcher to validate their own work. The research contribution of testing papers can be improved using controlled experiments with a standard benchmark, especially when considering the performance prediction techniques that in general are not validated against other techniques. As noted in Subsection II.4.1, performance prediction techniques employ a lot of different characteristics/parameters of the input dataset, program functionality, programming cluster and file system. In consequence, there is no clear intuition of which parameters have more influence in performance. The researchers can improve the testing techniques with both an accurate analysis of the parameters that have more impact in performance and rigorous experimentation using other testing techniques as a baseline.

This chapter analyzed 54 studies in detail, obtained through a wide search that resulted in 1377 *Big Data* studies by applying a filter (C4), in which only the studies that address software testing of *MapReduce* applications pass. Of these 1377 *Big Data* studies, 1043 are about *Big Data Engineering* and 334 about *Big Data Analytics*. Table 15 classifies the *Big Data Engineering* studies based on the research topic in order to characterize the research efforts. This classification reflects the research efforts to boost the *Big Data Engineering* field because 44.1% of the studies improve the technology, 18.31% analyze the technology through studies and surveys, 9.01% create new technologies to manage and analyze data, and 6.62% are focused on the state-of-the-art and challenges. Despite the challenges of testing in the *Big Data* area [108], [110], there are few research lines which focus on testing *Big Data* programs in general and *MapReduce* programs in particular.

The most relevant findings of this *mapping study* are enumerated in Table 16 and discussed in the next Section.

Table 15 Number of Big Data engineering studies in the last filter of the mapping study

		Number of studies		
Improvements of technology	**Performance**	121 (11.6%)	460 (44.1%)	1043 (100%)
	Security	81 (7.77%)		
	Data acquisition, storage and extraction	45 (4.31%)		
	Fault tolerance and availability	42 (4.03%)		
	Energy	42 (4.03%)		
	Improvements outside of Hadoop	35 (3.36%)		
	Scheduling	34 (3.26%)		
	MapReduce model	14 (1.34%)		
	Different frameworks	10 (0.96%)		
	Other improvements	36 (3.45%)		
Studies/Surveys	**General quality in Big Data**	171 (16.4%)	191 (18.31%)	
	Other	20 (1.92%)		
Software testing	**For MapReduce programs**	64 (6.14%)	103 (9.88%)	
	For non-MapReduce programs	39 (3.74%)		
Big Data in the cloud		101 (9.68%)		
New frameworks	**New Hadoop frameworks**	85 (8.15%)	94 (9.01%)	
	Other new Frameworks	9 (0.86%)		
State-of-the-art and challenges		69 (6.62%)		
Debug		6 (0.58%)		
Other		19 (1.82%)		
Not applicable (Big Data Analytics)				334

II.5 Discussion of Results

This Section discusses the main findings obtained in the current *systematic mapping study* and enumerated in Table 16. Despite the recent interest in *Big Data* through several studies published to improve/study the underlying technology, few of them are focused on software testing [Finding 1]. Researchers not only have opportunities in software testing for *Big Data* programs, but also for *MapReduce* applications. Although *MapReduce* is one of the processing models most frequently used in *Big Data*, the programs are usually formed by the integration of a stack/pipe of different technologies. In contrast, the majority of research about software testing is only focused on the *Map/Reduce* code, without considering the code of other technologies of the *Big Data* stack [Finding 2]. The testing techniques are usually similar to those employed in general purpose software, and so the researchers should adapt other general testing research to *MapReduce* considering the specific characteristics of the processing model.

The majority of studies about software testing in *MapReduce* applications are focused on performance using verification and validation test activities such as simulation or evaluation [Finding 3]. These tests are usually done to predict/forecast/analyze performance through models that use several parameters characterizing both the program functionality, the programming cluster and the file system [Finding 4]. Since each of these models employs

Table 16 Findings of the mapping study

Id	Finding
1	Despite several studies that are aimed at both improving and studying the state-of-art of *Big Data* technology, there are in comparison few research lines focused on software testing of the *Big Data* programs [Subsection II.4.3]
2	The majority of testing research in *MapReduce* applications is focused on either *Map* or *Reduce* or the integration of both, and cannot be applied to other processing models because they are specifically designed for *MapReduce* [Subsections II.4.2.3 and II.4.2.4]
3	The majority of research is about performance testing, and, to a lesser degree, functional testing [Subsection II.4.2.2]. This research is about verification and validation analysis, and, to a lesser degree, about dynamic testing [Subsection II.4.2.3]
4	The prediction/analysis models employed in performance testing use different numbers of heterogeneous parameters based not only on the *MapReduce* program functionality, but also on the cluster infrastructure, file system and data [Subsection II.4.1.1]
5	The most frequent reasons for testing the *MapReduce* programs are based on performance issues (analyze, optimize and fulfill performance goals), existence of several and specific failures, the type and quality of the data processed by these programs, and testing to predict and efficiently select the resources [Subsection II.4.2.1]
6	There are several rigorous reasons for testing the functionality of *MapReduce* applications, such as the percentage of programs that fail in production or the improper use of both functional semantics and data, but there are not many research efforts focused on this line of interest [Subsection II.4.3]
7	Whereas performance testing is done by simulation and evaluation, functional testing employs different test activities, such as static testing and structure-based testing [Subsection II.4.3]
8	As expected, testing research is focused on the software qualification process to help the tester [Subsection II.4.2.4]
9	The majority of research neither creates nor uses a tool for testing *MapReduce* programs [Subsection II.4.2.3]
10	Software testing research focused on *MapReduce* applications is usually published in conferences, and furthermore it is usually published without a strong validation, using only some case studies instead of rigorous empirical experiments [Subsection II.4.3]

different heterogeneous parameters, then it is difficult to understand which are the ones that really affect performance, as well as the real weight/influence that these parameters have in performance. Performance testing research could be improved by means of both an analysis of the parameters used by other researchers, and rigorous experimentation using other models as a baseline.

According to the research lines, the main reason for testing *MapReduce* applications is performance [Finding 5]. Also the majority of the testing techniques for *MapReduce* applications are related to performance, as expected. The research lines also suggest that functionality is another of the relevant reasons to test *MapReduce* applications, but the actual number of functional testing techniques is low [Finding 6]. Researchers may have opportunities to devise new functional testing techniques considering the specific characteristics of *MapReduce* programs such as distributed execution and scalability, among others. The functional testing techniques of *MapReduce* programs involve different test activities, which include structure-based, static analysis and formal methods [Finding 7]. The researchers should adapt the dynamic/static/formal testing techniques of general-purpose software (data-flow, combinatorial or mutation testing, among others) to *MapReduce* considering the specific characteristics of the programming model. In the current thesis, Section I describes a functional testing technique for *MapReduce* programs based on Random testing [155], and Partition testing [156] together with Combinatorial testing [157], [158].

Regardless of performance or functionality, the majority of testing is aimed at helping the tester in the software qualification process [Finding 8] without tools [Finding 9]. The contributions of researchers could not only help the testers, but could also help the final users providing automatic tools to support the design of test cases, and monitoring tools to analyze failures produced at runtime in production. In the current thesis, Chapter III describes a technique to detect faults for *MapReduce* programs during the runtime in production, and Chapter IV support their debugging.

The majority of studies about software testing in *MapReduce* applications are published in conferences and evaluated with some case studies [Finding 10]. Researchers could improve both visibility and quality by means of rigorous experiments based on a benchmark of *MapReduce* programs that can expose functional/performance failures, such as SWIM [159], GridMix [160], SparkBench [161], BigBench [162] or TPCx-BB [163]. The current thesis is focused on design faults, but there is no a known benchmark for *MapReduce* programs that contains design faults. Then the thesis is evaluated with real-world programs obtained from both Internet and organizations that are interested in the quality of their *MapReduce* programs.

II.6 Limitations of the Mapping Study

Despite the fact that both the planning and the execution (conducting) of this *mapping study* aimed to avoid bias, some limitations and researcher decision biases could exist [164].

- The results are limited by the academic context because the data sources are focused on the research field. Bias could be generated if the research papers do not represent the reality and motivations of software testing in *MapReduce* programs.
- Following some practices from social science [99] and software engineering [100], the selection of the primary studies was performed by one author for those papers that are clearly non-relevant. In contrast, two authors selected the primary studies independently from 1043 studies that had more chances to be relevant. Both authors agreed in 96% of studies and obtained a substantial/moderate agreement with 0.69 as a Kappa coefficient and [0.60-0.78] as 95% confidence interval.
- Despite the authors not finding quality problems in the primary studies, the quality of these studies was not formally evaluated. The same issue occurs in the majority of the *systematic mapping studies* [100] because quality assessment is usually not required [33].
- The data extraction was performed by one author and checked by another author. This practice is used in other *systematic reviews* [106] and some researchers consider it more practical than when data is extracted by several authors [98].
- Further bias occurs if some research questions cannot be properly answered through the checklist of the data extraction. In order to minimize this bias, the majority of these checklists are based on the international standards.

- Another less important potential bias could occur during the search process if some primary studies are not found with the search terms or expert opinions. In order to minimize bias, a thorough search is performed in several databases, journals, conferences and experts.

In order to avoid bias in the results, all steps are reviewed and some countermeasures are taken in research questions, the search process, data extraction and data analysis:

1. Research question: created by the Kipling method [70] instead of ad-hoc.
2. Search process:

   Search terms: the use of a large number of terms could improve the search process by obtaining more potential primary studies. Some authors encourage the use of several short queries instead of long queries [165]. This *mapping study* searches for a combination of 92 *MapReduce* related terms and 102 testing terms obtained from ISO/IEC 25010:2011 Quality (sub)characteristics [72] 44 with synonyms obtained through Kitchenham et al. [67] points of view.

   Data sources: this study searched 5 electronic databases recommended by Kitchenham et al. [83] and 2311 proceedings/volumes related to software testing of *MapReduce* programs. The other data source taken into account is the opinions of experts in the field in order to minimize the bias by adding primary studies that could not be found by the previous search.

   Study selection: this *mapping study* excludes non-relevant studies based on 4 filters. These filters were reviewed in order to obtain the relevant studies.
3. Data extraction: the majority of the data extracted are based on checklists, in some cases obtained from international standards and in others created or adapted to the *MapReduce* processing model.
4. Data analysis: the methods used in this chapter are employed in software engineering [92].

II.7 SUMMARY

The number of studies on software testing of *MapReduce* programs has increased during recent years. A characterization was carried out based on 54 research studies obtained from more than 70000 potential papers. The testing tasks in these programs are normally performed by the tester in the Software/System Qualification Testing Process due to a combination of the following 7 reasons: performance issues, potential failures, issues related to the data such as for example data quality, the reduction of the cost in resources, misconfigurations, improper use of the technology, time problems or other issues. These reasons for testing assume that both functional and performance testing are necessary, but the studies employ different approaches: functional testing considers different aspects of the program (such as specification and structure) while performance testing is more focused on simulation and evaluation. The current body of research focuses on performance testing, while there is a challenge in functional testing due to the importance of this line of research and the lack of research efforts. The current thesis is motivated on this challenge and focused on the functional faults caused by the wrong design of *MapReduce* programs.

The main goal of performance testing in *MapReduce* studies is to predict the execution time and the resources required to efficiently execute the programs and satisfy the agreements. From the functionality point of view, the goal of the studies is to detect faults considering the specific characteristics of the *MapReduce* processing model. Regardless of the type of testing, the majority of efforts are specific for the *MapReduce* technology at unit and integration level of the *Map* and *Reduce* functions. This situation may indicate a challenge in the integration of *MapReduce* programs with other programs, especially other *Big Data* stack technologies.

The research into software testing in *MapReduce* programs is mainly validated with example programs. There is scope to evolve with better validations and thus improve the research impact. Despite the lack of maturity, several studies create tools to support testing, but few are available on the Internet for users or other researchers. In *Big Data* there are few research studies related to software testing in comparison to the number of research efforts focused on improving the technology, which indicates new opportunities in software testing of *Big Data* in general, and *MapReduce* in particular.

The next chapters of the thesis are focused on both software testing and debugging of functional faults of the *MapReduce* programs caused by wrong design. The techniques proposed are automatized in tools and also evaluated through controlled experiments.

III TESTING

The *MapReduce* programs can have different kind of functional faults. A study of 507 programs in production reveals at least 5 different kinds of faults that are caused by a wrong design of *MapReduce* programs [29]. Other researchers identify and classify more of such design faults of the *MapReduce* applications [38], [143]. In this chapter we propose new testing techniques to address these functional faults that are caused by incorrect design. These types of faults include, but are not limited to, race conditions, computations with unavailable data because the distributed system allocates them to another computer, or automatic optimizations that remove data that are relevant to calculating the output. These faults are difficult to detect because they depend not only on the data, but also on how these data are executed in the large distributed architecture (infrastructure configuration): parallel executions, re-executions of some part of the data and optimizations, among others. In general, these non-deterministic faults are easy to mask in development/testing environments and go on to fail in more complex environments such as the production environment, thereby generating incorrect outputs or causing the program to crash.

In order to detect the design faults of the *MapReduce* programs, this chapter proposes a testing technique that executes the test case under the relevant and representative infrastructure configurations. The majority of this chapter is published in IEEE TR 2018 [41]. Section III.1 introduces the faults of the *MapReduce* applications caused by a wrong design. Related work is then discussed in Section III.2. The testing techniques proposed and the automatization (MRTest) are defined in Section III.3. The experiment is performed and discussed in Section III.4. Finally, the conclusions and future work are detailed in Section III.5.

III.1 BACKGROUND OF DESIGN FAULTS IN MAPREDUCE

MapReduce programs process large datasets distributed over several computers using the "divide and conquer" principle. In its simplest form, the *MapReduce* developer needs to implement only two components: the *Mapper* that splits one problem into several subproblems (Divide), and the *Reducer* that solves these subproblems (Conquer). During the execution, several instances of *Mapper* analyse the dataset in parallel and send to each subproblem the data needed to be solved. After all *Mappers* are executed, several instances of the *Reducer* are executed in parallel to solve the subproblems. Internally, the data are codified as <key, value> pairs, where the key is an identifier of a subproblem, and the value contains all the information that is needed to solve the subproblem. The developer designs the business logic based on the <key, value> pairs emitted from *Mappers* to *Reducers*. Finally, the output is a series of <key, value> pairs obtained through the deployment and execution of *Mappers* and *Reducers* over a distributed infrastructure.

More generally, a *MapReduce* program can be designed with more components. For example, a *Combiner* can be implemented to improve the performance by reducing the data exchanged between *Mappers* and *Reducers*. The *Combiner* is executed right after the *Mapper* with the aim of removing the <key, value> pairs that are irrelevant to solving the subproblem. The *MapReduce* applications can also be designed with other components such as, for example, the *Partitioner* that determines which *Reducer* analyses which <key, value> pair, the *Sorter* that controls the order of <key, value> pairs, and the *Grouper* that aggregates the values of each key before they are passed to the *Reducer*.

Distributed systems such as Hadoop execute the *MapReduce* programs in a non-deterministic way based on runtime factors, such as the resources available, observed infrastructure failures and other dynamic optimizations. Nevertheless, the same program with the same input data when executed in different infrastructure configurations should obtain the correct output. However, this is not always the case: in our first work in the second line of research [38] we identified and classified several design faults that are raised in some infrastructure configurations but masked in others. Despite the fact that some authors suggest that the parallel programming must be deterministic by default (unless the developer explicitly indicates non-determinism) [166], this is not the case with these distributed systems.

To illustrate *MapReduce* and its executions, let us suppose the computation of the average temperature per year given a large dataset containing several years with their observed temperatures. This program can be designed in different ways. We suppose that the developer makes the following decisions. The problem of average temperature per year is divided into several subproblems where each subproblem calculates the average temperature of one year only (Decision 1). Then each subproblem is composed of one year with all temperatures of this year (Decision 2), and it is solved with the temperature average (Decision 3). The program includes a *Combiner* to improve the performance (Decision 4). With the foregoing decisions, the *Mappers* receive a subset of temperature data and emit <year, temperature of this year> pairs. Then the distributed system aggregates all values per key, that is, each subproblem grouped with all the data that needs to be solved. Therefore, the *Reducers* receive subproblems like <year, [all temperatures of this year]> and calculate the average temperature. After the *Mapper*, the *Combiner* can be executed, aimed at removing the irrelevant temperatures, and emitting their average.

The distributed system can execute the previous program in different ways, based on the runtime infrastructure configuration. For example, Fig. 10 shows three different executions with the following input: year 1999 with 4°, 2° and 3°; and year 2000 with 5°. Regardless of the infrastructure configuration, the program must obtain the right output: 3° as average in 1999, and 5° in 2000. The first configuration is the simplest with one *Mapper*, one *Combiner* and one *Reducer*. The *Mapper* analyses all temperatures and encodes them as <year, temperature>. Then the temperatures are grouped per year and sent to the *Combiner* that pre-calculates the average temperatures, and finally to the *Reducer* that obtains the correct output.

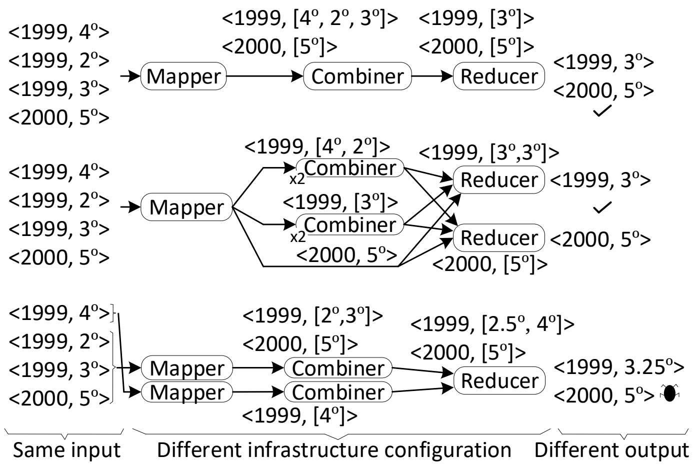

Fig. 10 Execution of MapReduce program that calculates the average temperature per year

Depending on the runtime resources, the distributed system can execute the program automatically in more complex configurations. As we detail in Section IV, the configurations can have, among other things, a different number of *Mappers* and other automatic optimizations. For example, the second configuration of Fig. 10 is more complex than the first, employing one *Mapper*, two *Combiners* and two *Reducers*, also obtaining the correct output. We assume that the first *Combiner* receives the temperatures 4° and 2° of the year 1999, and emits their average, 3°. The second *Combiner* receives year 1999 with 3° and emits it, whereas the year 2000 with 5° is passed directly from *Mapper* to *Reducer*. After the *Mappers* and *Combiners* are executed, one *Reducer* analyses the temperatures of the year 1999 and another *Reducer* the year 2000. Eventually, this configuration also obtains the correct output.

In contrast, the third configuration that executes two *Mappers*, one *Combiner* per *Mapper*, and one *Reducer* does not obtain the right output. This execution obtains 3.25° as the average of 1999 rather than 3° due to a design fault. The developer makes some incorrect design decisions, among them, the use of <year, temperature> pairs and the *Combiner* to optimize the program. Both decisions are incompatible in this program because the *Combiner* replaces the temperatures locally available in each computer with their average, and then the *Reducer* cannot calculate the global average using only the local averages.

Although this program has a simple business logic, several developers make the previous incorrect design decisions to obtain the average temperature per year, as in the programs [167], [168]. The developer can fix the program by removing the *Combiner*, but this solution is not optimised. A better program design codifies the data as <year, {sum of temperatures, number of temperatures}> and uses a *Combiner* to update both the sum and number of temperatures [169].

A sample of a more subtle design fault is in the recommendation system Open Ankus [170]. The users assign points to a series of books, and then the system tries to forecast the points assigned for new books. The design fault is triggered during the calculation of the error between the user assignment and the system prediction. Fig. 11 depicts the execution of the program with the points assigned by Alice to the book Don Quixote and correctly predicted by the system. The first configuration with one *Mapper* for the predictions and one *Mapper* for the assignments obtains the correct output (the system predicts the result correctly). In contrast, when several *Mappers* for assignment are executed in parallel, the output of this program could potentially be faulty, depending on both the order of execution and how the data is distributed in parallel. For example, the program can be executed as in the second configuration of Fig. 11 obtaining the incorrect result that the system prediction is wrong.

When the business logic tends to be more complex, as in machine learning programs, it can be difficult for the developer to make the right design decisions, and the program may be prone to side-effects. An incorrect design in the *MapReduce* program may cause a failure in one of the different ways in which the distributed systems can execute the program. These design faults are difficult to detect during testing because they may depend on dynamic execution contexts. Thus they can be missed in the laboratory, but are then triggered in aggressive environments, such as a production environment with a mix of large data and infrastructure failures. When these aggressive situations happen, the distributed systems manage the execution with different mechanisms, such as re-executing part of the program or performing some optimizations that can reveal design faults. To avoid incorrect outputs in production, it is desirable to detect these program faults in the early stages of the development process.

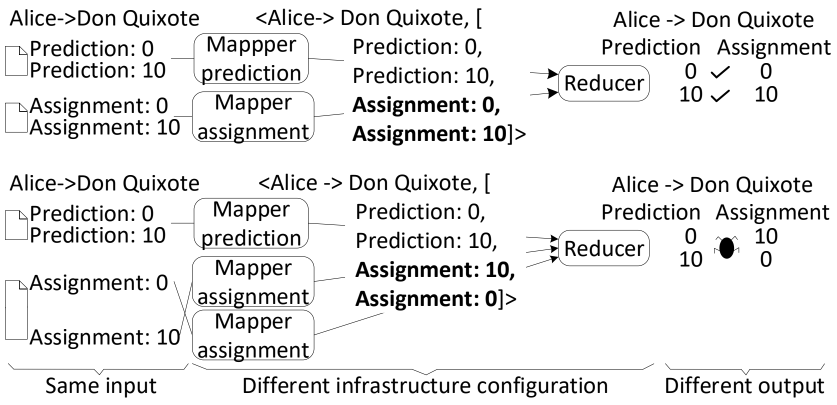

Fig. 11 Execution of a recommendation system in MapReduce

III.2 Related Work

Software testing is among the most commonly used software quality-assurance techniques [171]. In recent years, this field has seen great progress [31], but concerning the testing of *Big Data* applications, there remain several challenges according to the previous chapter. In this domain, most works focus on performance testing, but, functional testing is also important to avoid incorrect outputs. In this chapter we address functional testing.

As seen in the earlier examples, some faults depend on how distributed systems execute the programs according to the infrastructure configurations. If the program generates incorrect outputs in some configurations and the expected output in others, then the program has a design fault. A study of 507 *MapReduce* programs in production reveals at least 5 different kinds of design faults [29]. To detect them, Csallner et al. [145] and Chen et al. [144] use testing techniques based on symbolic execution and model checking. Other authors [38], [143] identified and classified other design faults that depend upon the infrastructure configurations. This chapter proposes a test approach to detect such faults in the test environment by using a simulation of the infrastructure configurations based on combinatorial strategies.

The production environment is composed of a large distributed infrastructure that over time exposes several failures [59]. In order to test in the same conditions as production, several research lines propose to inject infrastructure failures [85], [148] during testing, and several tools have been implemented to support their injection [64]–[66]. For example, Marynowski et al. [147] propose creating the test cases by specifying which computers fail and when. While some of the design faults can be detected by injecting infrastructure failures, others require a fine-grained control of the distributed system and the underlying large infrastructure. In the production environment it is difficult to control the execution of the test cases because at the same time other programs consume the resources of the computers and other infrastructure failures can happen that are beyond the tester’s control. This chapter does not inject failures, but simulates the different infrastructure configurations in a test environment, thereby obtaining fine-grained control and reproducibility of the tests.

Several research lines propose generating the test input data through different approaches: using a bacteriological algorithm [149], or with input domain analysis together with combinatorial testing [172]. Unlike these testing techniques, this chapter does not focus on the generation of the test input data, but on simulating their execution in the infrastructure configurations that are more likely to reveal the faults. As such, the technique of this chapter is orthogonal to the above. The tester can use the previous approaches to obtain the test input data and then execute the test cases with the techniques proposed in this chapter. As we have shown, the same program and the same input data executed in different configurations might produce different results so, apart from deriving a good test suite, the testing of *MapReduce* applications also requires the derivation of the correct configuration.

Several tools have been proposed to design and execute test cases for *MapReduce* applications. Herriot [63] allows the execution of the tests in a distributed infrastructure and at the same time supports the injection of infrastructure failures. Another tool called MiniClusters [62] executes the test cases in a distributed environment simulated in memory. For unit testing, MRUnit [61] provides an adaptation of JUnit [60] to the *MapReduce* processing model. All the above test tools only execute the test case in one infrastructure configuration and usually without parallelization. In this chapter we devise a testing technique to generate and execute a representative set of infrastructure configurations that could occur in production and as a whole is more likely to reveal design faults. It is automated by means of an MRUnit extension, as described below.

III.3 MRTEST: AUTOMATIC MAPREDUCE TESTING TECHNIQUE

In this section, we describe the test execution engine we propose called MRTest. Given a test input data, MRTest automatically generates the configurations aimed at revealing design faults (Subsection III.3.1), then executes the test case in these configurations (Subsection III.3.2), and finally checks if the program is faulty or not (Subsection III.3.3).

III.3.1 Generation of Infrastructure Configurations for Testing

This chapter proposes an automatic technique that, given test input data, generates a different number of configurations to test the programs. Then the faults are revealed when a failure occurs in one of these configurations. Ideally, these faults are detected executing all possible configurations, and as initial work we proposed to generate these thorough number of configurations (MRTest-Thorough), but the approach exposed limitations because it takes a long time and only supports a small volume of test input data [42]. In order to overcome these problems, the thesis proposes other two techniques reducing the number of configurations generated while maintaining the fault detection effectiveness: the two techniques use Random Testing [155] (MRTest-Random) and Partition Testing (Equivalence Partitioning [156] with Combinatorial Testing [157], [158]) (MRTest-t-Wise).

The first technique, **MRTest-Random**, generates the configurations randomly from all valid configurations. The tester indicates the number of configurations wanted, and MRTest-Random generates them randomly.

The second technique, **MRTest-t-Wise**, divides the set of all valid configurations into several partitions with similar behaviour and applies a combinatorial strategy to generate the configurations under test. In software testing, depending on the failure probability [173], Random testing can be as effective as Partition testing [174] and can be a feasible option [175]. In other circumstances, Partition Testing can be more effective than Random Testing [176]. As we discuss in Section V, our experiments show that both techniques MRTest-Random and MRTest-t-Wise can be effective in revealing *MapReduce* faults, but MRTest-t-Wise is significantly better. The latter testing technique is schematically represented in Fig. 12 and described below

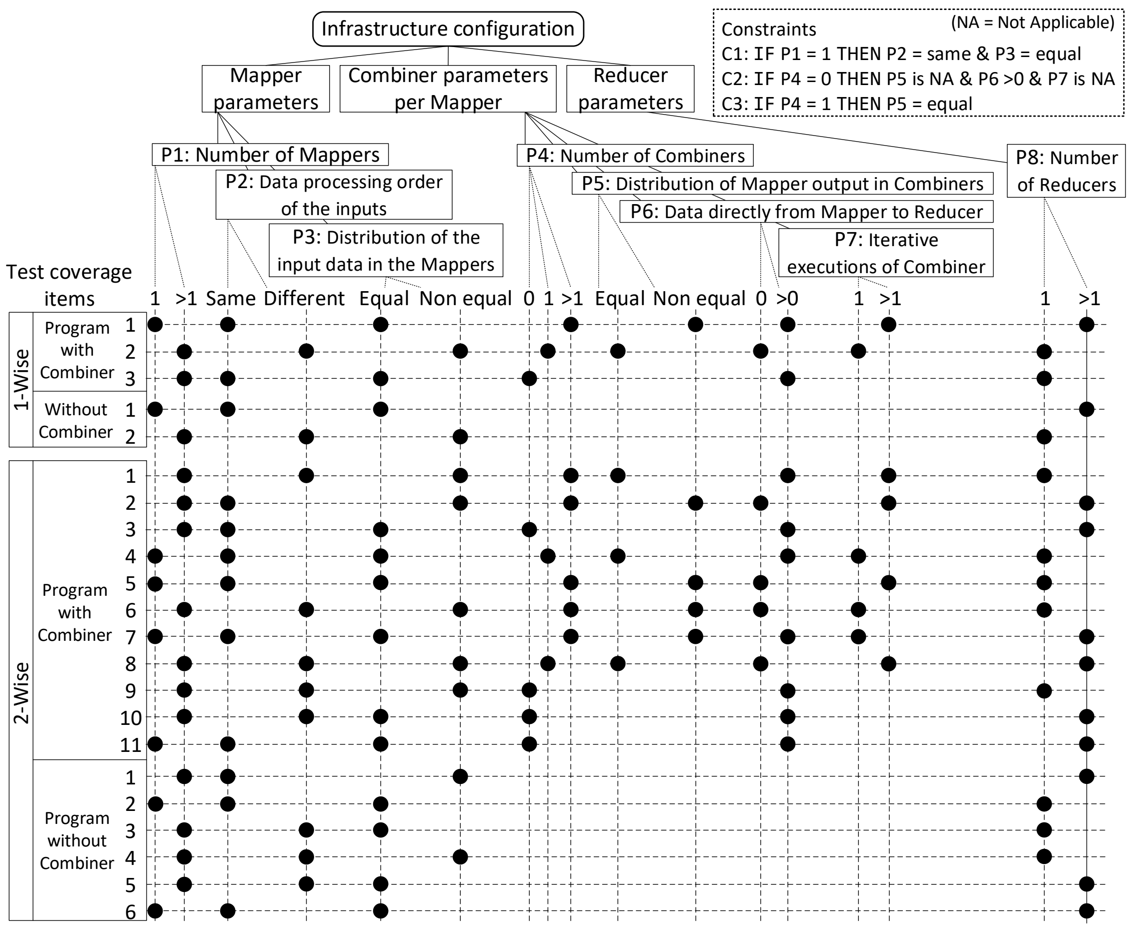
Infrastructure configuration
Mapper parameters
Combiner parameters per Mapper
Reducer parameters
Constraints
(NA = Not Applicable)
C1: IF P1 = 1 THEN P2 = same & P3 = equal
C2: IF P4 = 0 THEN P5 is NA & P6 >0 & P7 is NA
C3: IF P4 = 1 THEN P5 = equal
P1: Number of Mappers
P2: Data processing order of the inputs
P3: Distribution of the input data in the Mappers
P4: Number of Combiners
P5: Distribution of Mapper output in Combiners
P6: Data directly from Mapper to Reducer
P7: Iterative executions of Combiner
P8: Number of Reducers
Test coverage items
1
>1
Same
Different
Equal
Non equal
0
1
>1
Equal
Non equal
0
>0
1
>1
1
>1
1-Wise
Program with Combiner
Without Combiner
2-Wise
Program with Combiner
Program without Combiner

Fig. 12 MRTest-t-Wise testing technique

in three parts: (1) Division of the set of all valid configurations, (2) Combination strategy, and (3) Generation of the configurations.

Division of the set of all valid configurations: The set of all valid configurations is divided based upon the following parameters and constraints that are also represented at the top of Fig. 12:

Mapper:

P1) Number of *Mappers*: 1 or >1. The program in production can be executed with one *Mapper* (1) that analyses the entire dataset, or alternatively with several *Mappers* that analyse different parts of the dataset in parallel (>1).

P2) Data processing order of the inputs: data are processed in the same order as they are encountered in the input (same), or in a different order (different). The *MapReduce* processing model does not guarantee that the data will be processed in the same order as they are stored in the input.

P3) Distribution of the input data in the *Mappers*: data equally distributed in the *Mappers* (equal) or not equally distributed (non equal). The *Mappers* process different subsets of input data: there could be configurations with an equal number of data in each *Mapper*, or with a different number of data.

Combiner:

P4) Number of *Combiners* per *Mapper*: 0, 1 or >1. Each *Mapper* can execute one *Combiner* (1), several *Combiners* (>1) or can emit the data directly to *Reducer* (0).

P5) Distribution of *Mapper* output in *Combiners*: data equally distributed in *Combiners* (equal) or not equally distributed (non equal).

P6) Data directly from *Mapper* to *Reducer*: 0 or >0. All data emitted by *Mappers* can be pre-processed by the *Combiner* functionality (0), or in contrast some data can pass directly from *Mapper* to *Reducer* without executing the *Combiner* functionality (>0).

P7) Iterative executions of *Combiner*: 1 or >1. The output of the *Combiner* can be executed iteratively by the *Combiner* several times (>1) or only once (1).

Reducer:

P8) Number of *Reducers*: 1 or >1. The program can be executed in production with one *Reducer* that solves all subproblems (1) or with several *Reducers* that solve the subproblems in parallel (>1).

For example, the configuration at the bottom of Fig. 10 is characterized by the following parameters:

- P1 is >1: There are two *Mappers*.
- P2 indicates a different order: The input data is executed in a different order than they are stored in the input. The 4° temperature is executed after 2°, but in the input the temperature 4° is before 2°.
- P3 indicates a non-equal distribution of the data in *Mappers*: Each *Mapper* has a different number of input data. One *Mapper* has 1 register and the second has 3 registers.
- P4 is 1: Each *Mapper* only executes one *Combiner*.
- P5 indicates an equal distribution of the *Mapper* output in its *Combiners*. Each *Mapper* only has one *Combiner* that receives all its data, then the output of the *Mapper* is equally distributed in its *Combiner*.
- P6 is 0: There are no data that pass directly from the *Mapper* to the *Reducer* without the *Combiner*.
- P7 is 1: The *Combiners* are not executed iteratively several times, they are executed only once.
- P8 is 1: There is only one *Reducer*.

The configurations under test are obtained by a combination of the previous parameters. However not all combinations make sense, and to prevent non-meaningful combinations, we have derived the following constraints that descend from the *MapReduce* processing model:

- When the number of *Mappers* (P1) is 1, then: (a) Data processing order of the inputs (P2) is the same order as they are in the input, and (b) Distribution of the input data in the *Mappers* (P3) is equally distributed.
- When the number of *Combiners* (P4) is 0, then: (a) the distribution of the data in the *Combiners* (P5) is not applicable, (b) the Data directly from the *Mapper* to the *Reducer* (P6) is >0, and (c) the iterative executions of the *Combiner* (P7) is not applicable.
- When the number of *Combiners* (P4) is 1, then the data in the *Combiners* (P5) are equally distributed.

Combination strategy: Deriving all possible combinations of previous parameters is expensive and the t-Wise strategy (also known as t-Way) is applied [88], [157]. Instead of combining all the values of all parameters, t-Wise [177] combines only the values of all subsets of t parameters. For example, 1-Wise (each use) [178] requires that all values of each parameter appear in at least one test case, whereas 2-Wise (pairwise) requires that the combination of all values per pair of parameters appears in at least one test case. 2-Wise has been shown to be almost as good as all combinations of parameters [179] at detecting failures, but employing much fewer resources in terms of time and cost [180].

The MRTest-t-Wise technique generates the configurations covering the t-Wise combinations of the previous parameters and constraints. Fig. 12 details the configurations that must be covered (test coverage items) for 1-Wise and 2-Wise strategies. Each row of the figure represents a test coverage item that should be covered with a configuration that satisfies the parameters indicated by dots. The 1-Wise technique requires the generation of 3 configurations (test coverage items) when the program implements a *Combiner*, and 2 configurations in the other case. The 2-Wise is a more thorough combination, requiring 11 configurations for programs with a *Combiner*, and 6 when the program does not implement a *Combiner*.

Generation of configurations: The configurations can be created manually to cover each test coverage item of the t-Wise selected, but the MRTest-t-Wise technique generates these configurations automatically. The following pseudo-code describes how the configurations are generated:

```
Input:  t-Wise (testing technique selected, i.e. 2-Wise)
        sut (software under test)
Output: Configurations that cover t-Wise in sut
(1)   Configurations ← ∅
(2)   tcis ← Get all test coverage items of the t-Wise
(3)   ∀ tci ∈ tcis
(4)   |     Configuration ← ∅
(5)   |     ∀ parameterToCover ∈ tci
(6)   |     |     value ← obtain randomly a value that covers
                                          parameterToCover in sut
(7)   |     |     IF value exists:
(8)   |     |           Configuration ← Configuration ∪ value
(9)   |     |     ELSE  //When there is no value to cover
                                                 parameterToCover
(10)  |     |           The actual configuration cannot cover the  test
                        coverage item in sut, then backtracks trying to
      |     |           generate  again the  configuration  with  other
      |     |           values in previous parameters
      |     |_          [maximum τ times (threshold)]
(11)  |     IF Configuration covers the tci in sut:
(12)  |_          Add Configuration to Configurations
(13)  RETURN Configurations
```

In order to generate a configuration that covers each one of the test coverage items (1, 2, 3), the MRTest-t-Wise covers the first parameter with random values, then the second, and so on (4, 5, 6, 7, 8). For example, if the first parameter should be P1: >1, i.e., more *Mappers*, then a random value is selected greater or equal to 2 to guarantee >1 *Mappers*, and so on with the remainder of the parameters. Sometimes it can be impossible to cover one parameter because

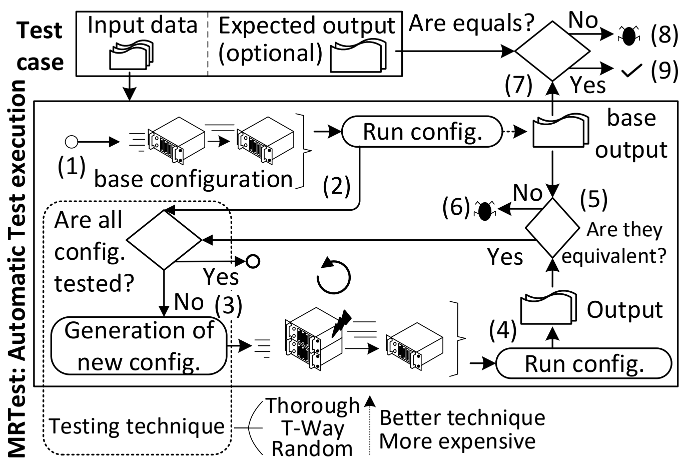

Fig. 13 MRTest test execution engine

the test input data add semantic constraints unknown until the values selected in the previous parameters are executed (9, 10). For example, sometimes it is impossible to obtain a configuration with >1 *Reducers* because the test input data always lead to one *Reducer*. In these cases MRTest-t-Wise uses a backtracking approach. First it fulfils randomly the first parameter (4, 5, 6), then it executes the part of the program affected by this parameter (7, 8) and tries to cover the second, and so on. When the generator discovers at runtime that one parameter cannot be covered, then it backtracks, changing the value of previous parameters (9, 10). For example, a configuration can be created with 2 *Mappers* (P1 >1) but three input data cannot be equally distributed in them (it is impossible to cover P3 with equal distribution), then the generator backtracks changing the configuration to three *Mappers* (it maintains P1 >1 but changes randomly its value), and finally the three data items can be distributed equally in the *Mappers*. A threshold is set to prevent the generator from performing indefinitely or from backtracking for too long. When the threshold is overcome, then the technique does not create the configuration and the test coverage item is not covered (11, 3). By default, the threshold is 15 backtracks because we observed that usually when this number is exceeded then it is infeasible to cover the test coverage item, regardless of the set of valid configurations. Finally, those configurations that cover the test coverage items are generated (11, 12, 13).

III.3.2 Execution of Test Cases

In order to detect design faults, MRTest executes each test case in different configurations using one of the techniques described in the previous section (MRTest-Random, MRTest-t-Wise and MRTest-Thorough). Then MRTest checks systematically that all configurations lead to equivalent outputs.

Given a test case with input data and, optionally, the expected output, the MRTest test execution engine is described in Fig. 13. First, MRTest executes the test input data in the base configuration (1), that is the simplest configuration with one *Mapper*, one *Combiner* and one *Reducer* without parallelization. Next, new configurations are iteratively generated (2,3) and executed (4) for a given testing technique selected by the tester: MRTest-Thorough, MRTest-t-Wise or MRTest-Random. The output obtained executing each configuration is checked against the output of the base configuration (5), revealing a fault if these outputs are not equivalent (6). Then MRTest can reveal faults with only the test input data, but the tester can optionally declare the expected output. In this case, the output of the base configuration is also checked against the expected output (7), detecting a fault when both are not equivalent (8, 9).

For example, let us revisit the program in Subsection III.1 that calculates the average temperature per year. Fig. 1 describes the 1-Wise execution of a test case with the following test input data: year 1999 with 4°, 2° and 3°; and year 2000 with 5°. Firstly, MRTest generates and executes the base configuration (top of the figure) obtaining 3° as average in 1999, and 5° in 2000 (1, 2). Then MRTest generates and executes a configuration to cover the first test coverage item of 1-Wise (middle of figure), and again obtains the same output (3, 4, 5). In contrast, when it generates and executes the configuration of the third test coverage item (bottom of the figure), it obtains 3.25° as average of 1999 instead of the 3° obtained in the base configuration (3, 4, 5). Then MRTest automatically reveals a fault because the two outputs are different (6). We discuss further the oracle used in MRTest in the following subsection.

The test execution engine MRTest was implemented based on MRUnit library [61] maintaining its API and including new functions to indicate the testing technique to be used. This library is used to execute each configuration. In MRUnit, the test cases are executed with the base configuration, but this library is extended to generate other configurations and enable parallelism supporting the execution of several *Mapper*, *Combiner* and *Reducer* tasks. This test execution engine employs randomness to generate the configurations, but also supports pseudorandom numbers, also called seeds, to guarantee that the execution of the test case is reproducible in a deterministic way.

III.3.3 Test Oracle

In software testing, the mechanisms that determine if the test reveals a fault or not are called test oracles [181]. There are some properties that characterize the efficacy of the test oracles [182], [183]. As discussed, if a design fault is present, the same program executed under different configurations can lead to different outputs. Based on this observation, the MRTest execution engine can reveal faults automatically even without knowing the expected output. It employs an automatic partial-oracle [181] that is derived from the program executions [184] using metamorphic testing [45]. Given a test case (original test case), metamorphic testing generates new test cases varying the original test case (follow-up test cases) to detect faults in a relationship amongst them (metamorphic relationship).

According to the software testing standard [30], a test case not only uses the test input, but also other test data that specify requirements for the test, such as databases, or configuration in the case of *MapReduce* programs. MRTest intends to detect those design faults that not only depend on specific test input, but also on specific configurations. For these faults, the test case must be designed with both the test input and the configuration in mind. MRTest receives the test input and then the metamorphic testing is focused in the relationships between the potential configurations. Given the test input, MRTest executes these test input data on the base configuration (original test case). Then MRTest generates the follow-up test cases maintaining the original test input but, but providing each one with different configurations. Finally, MRTest checks that both original and follow-up test cases lead to an equivalent output (metamorphic relationship). Whereas the metamorphic testing techniques usually generate the follow-up test cases by varying the test input, our approach generates the follow-up test cases by maintaining the test input but varying the configuration of the system under test.

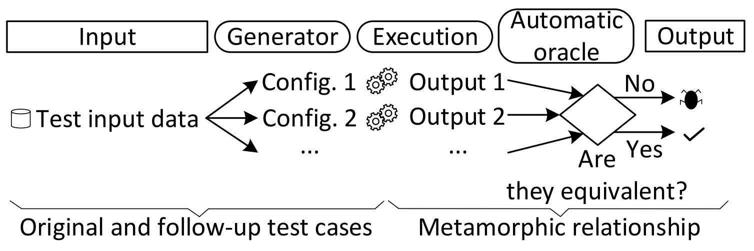

Fig. 14 Metamorphic oracle of MRTest

MRTest can also be employed when the expected output is previously unknown or costly to obtain, as occurs in several machine learning programs [185]. Fig. 14 describes how the MRTest oracle can detect faults when given only the input data. The original test case is the test input data executed in the base configuration (1 *Mapper*, 1 *Combiner*, 1 *Reducer*). Then MRTest generates and executes several configurations using the testing techniques described in the previous sections (follow-up test cases). Finally, it checks if their outputs are equivalent (metamorphic relationship), and if they are not then a potential fault is automatically detected.

According to the study of Segura et al. [186] the number of metamorphic papers will increase in years to come, but to date 49% employ the metamorphic testing capabilities in different problem domains, and only 2% present a tool. In our case, the testing technique of this chapter not only defines and automatizes the metamorphic relationship to the *MapReduce* domain, but it also develops a tool that detects faults easily with only the test input data.

III.4 Experiments

The **goal** of these experiments is the evaluation of how, using different configurations in the execution of the test cases, the effectiveness in failure detection could be improved without significantly decreasing efficiency. The approach proposed in this chapter, MRTest, executes the *MapReduce* test cases under several configurations, whereas the usual test execution engines, for example MRUnit, only execute them under a simple configuration. In the experiments, MRTest and MRUnit are compared in order to answer the following **research questions**:

RQ5. Do the test execution engines detect more failures when the *MapReduce* test cases are executed in different configurations?

RQ6. How expensive is the execution of the test cases in several different configurations?

The research questions are focused respectively on the effectiveness and efficiency during testing of the *MapReduce* programs. Depending on the field, the aptness of a technique can be referred to using different terms, for example: “effectiveness” for software testing techniques [187], “performance” for localization techniques [188], or “accuracy” for the classification techniques of machine learning [189]. In this chapter, we use the term “effectiveness” regarding the quantity of failures detected, and “efficiency” regarding the execution time employed by the techniques. The planning and the results of the related experiments are presented in the next two subsections, and the discussion of the experiments together with the limitations in Subsection III.4.3.

III.4.1 Effectiveness Experiments

The **goal** of the effectiveness experiments is the assessment of how many failures are detected. Following the Basili et al. [190] template, the goal is: *Analyze* the test case execution engines (MRUnit and MRTest) *for the purpose* of evaluation *with respect to* their respective effectiveness in detecting failures due to a program design fault against the *MapReduce* processing model *from the point of view of* the tester and developer *in the context of Big Data* applications. The **planning** of the experiments is described in Subsection III.4.1.1 and their **results** are reported in Subsection III.4.1.2.

III.4.1.1 Effectiveness: Setup

In this experiment, 8000 different test cases from 4 real world programs are executed in MRUnit and MRTest in order to analyse their capability to detect failures. Each one of the 4 programs has a known design fault that is only revealed in some of the potential configurations and masked in others. The programs used, including a summary of the functionality and the cause of the faults, are:

1. Open Ankus [170]: A recommendation system that predicts for each user the items that could be of interest to them (films, books, cities, and so on), based on choices of other users and their similarities to the user in question. This program could fail when the data of each user-item is split and parallelized.
2. Data quality analysis [191]: Measure of the quality of data interchanged between companies, based on international standards. This program did not correctly track the measurements and they could be incorrectly assigned due the parallel execution. The production version of this program has removed the fault.
3. Movie analysis [192]: Statistics analysis of movies, based on the ratings of users. This program is implemented with an incorrect *Combiner*.
4. Data cleaner Knn analysis [193]: Knn machine learning algorithm to clean text data, based on the number of transformations, insertions and removals of incorrect letters in the words of the text. This program fails when one *Mapper* needs data that are not locally available because it is assigned to another *Mapper*.

For each program, 2000 test cases that contain data able to trigger the faults are executed in MRUnit and MRTest with different modes. The test cases are generated iteratively with random data until we have 2000 test cases that are able to trigger the fault. Because the program faults are known, all of the potential test cases are automatically analysed in order to check if the data can generate incorrect outputs under at least one configuration of the *MapReduce* configurations. For example, the program described in Section III.1 that calculates the average temperature of each year, has a design fault that is not revealed by all inputs. We can automatically check if an input data is able to trigger the fault because the failure is only raised when the average of temperatures is different to the global average of local averages.

The **population** of the experiment is composed of all test cases with data able to trigger these faults in some configurations. Each of these test cases is then taken as the **experimentation unit**, and the **observation** is whether the test execution engines detect a failure or mask the fault. The **dependent variable** or response variable is the rate of failures detected by the different execution engines, which are the **independent variable**. The **baseline** is MRUnit and the **treatments** are MRTest executed in the following modes:

- 1-Wise: Based on the test coverage items proposed in MRTest-1-Wise algorithm, executes 3 or 2 configurations depending on whether the program has a *Combiner* or not, respectively.
- 2-Wise: Based on the test coverage items proposed in MRTest-2-Wise algorithm, executes 11 or 6 configurations depending on whether the program has a *Combiner* or not, respectively.
- 0-Random that executes randomly one configuration (MRTest-Random), in order to compare fairly with MRUnit that also executes one configuration (one *Mapper*, one *Combiner* and one *Reducer*).
- 1-Random in order compare fairly with 1-Wise. Executes 3 or 2 configurations depending if the program has a *Combiner* or not, respectively.
- 2-Random in order to compare fairly with 2-Wise. Executes 11 or 6 configurations depending if the program has a *Combiner* or not, respectively.

MRTest-thorough is not analysed in the experiments due to its limitations, such as only supporting a small amount of test input data or taking a long time to execute a test case. There are other elements that could affect the experiment and are treated as **blocking factors**:

- The size of the test input data could affect the rate of failures detected, so two sizes of data are considered: a small size (between 1 and 10 <key, value> pairs) and a larger size for functional testing purposes (between 11 and 35 <key, value> pairs).
- The generation of the configurations in MRTest is based on some pseudorandom functionality that could introduce noise in the failure rate. During the experiments, the different test engines employ the same pseudorandom number generated also in a pseudorandom way.

In the experiments two **sampling methods** are used: consecutive sampling to select the *MapReduce* programs and random sampling to select the test cases. Ideally the subject programs should be selected randomly, but as in the case in many software engineering experiments, this is not viable [194]. As such 4 real world programs that contain a known fault are selected instead.

As stated above, for each one of these programs, 2000 test cases that contain data able to trigger the fault are generated randomly. We grouped them in **trials** of 100 test suites with 20 test cases each: 50 test suites contain test cases with input data between 1 and 10 <key, value> pairs, and the other 50 test suites between 11 and 35 <key, value> pairs due to a pre-established blocking factor. All of these test suites are executed in the baseline (MRUnit) and the five treatments (MRTest), and then the rate of the failures detected is observed. This type of experiment design is called “within subject design with post-test”.

In these experiments, the **effectiveness** is measured by the percentage of failures detected per test suite. Then the effectiveness of the execution engines is compared via the **statistic test** Wilcoxon Sign Rank Test. This non-parametric statistic test analyses if there are significant differences based on the medians, then the null **hypothesis** is defined as H_{00}: $median(Effectiveness)_{MRUnit} = median(Effectiveness)_{MRTest}$

III.4.1.2 Effectiveness: Results and Discussion

Table 17 summarizes the number of test cases which detect a failure by each test execution engine (MRUnit and MRTest) during the experiments. This table shows that design faults against the *MapReduce* processing model are not detected in general by MRUnit, whereas MRTest approaches are able to detect them. The number of test executions that detect a failure by MRUnit is almost 0% whereas even considering the weakest MRTest approach, 0-Random, more than 15% of the test cases detect a failure; the strongest MRTest approach, 2-Wise, catches a failure in more than 60% of test cases. In general terms, 1-Wise and 1-Random detect more failures than MRUnit, and finally 2-Wise and 2-Random detect the majority of failures, regardless of the number of <key, value> pairs in the input data. In all approaches, the execution time of

Table 17 Effectiveness of failure detection of 100 test suites of 20 test cases for each one of the 4 real world programs with fault

		[1-10] <key, value> pairs		**[11-35] <key, value> pairs**		**Total**	
Program	**Treatment**	**Number of test cases that detect a failure**	**Effectiveness**	**Number of test cases that detect a failure**	**Effectiveness**	**Number of test cases that detect a failure**	**Effectiveness**
Open Ankus	MRUnit baseline	0	0.00	0	0.00	0	0.00
	0-Random	307	0.30	293	0.30	600	0.30
	1-Wise	490	0.50	302	0.30	792	0.40
	1-Random	513	0.50	479	0.50	992	0.50
	2-Wise	754	0.75	620	0.60	1374	0.70
	2-Random	898	0.90	861	0.85	1759	0.90
Data quality analysis	MRUnit baseline	1	0.00	3	0.00	4	0.00
	0-Random	773	0.75	900	0.90	1673	0.85
	1-Wise	816	0.80	954	0.95	1770	0.90
	1-Random	925	0.95	984	1.00	1909	0.95
	2-Wise	994	1.00	1000	1.00	1994	1.00
	2-Random	992	1.00	1000	1.00	1992	1.00
Movie analysis	MRUnit baseline	0	0.00	0	0.00	0	0.00
	0-Random	169	0.15	183	0.20	352	0.15
	1-Wise	562	0.55	395	0.40	957	0.48
	1-Random	378	0.35	328	0.30	706	0.35
	2-Wise	952	0.95	861	0.85	1813	0.90
	2-Random	694	0.70	534	0.53	1228	0.63
Data cleaner Knn analysis	MRUnit baseline	0	0.00	0	0.00	0	0.00
	0-Random	876	0.75	946	0.95	1822	0.90
	1-Wise	983	0.80	1000	1.00	1983	1.00
	1-Random	978	0.95	997	1.00	1975	1.00
	2-Wise	1000	1.00	1000	1.00	2000	1.00
	2-Random	1000	1.00	1000	1.00	2000	1.00

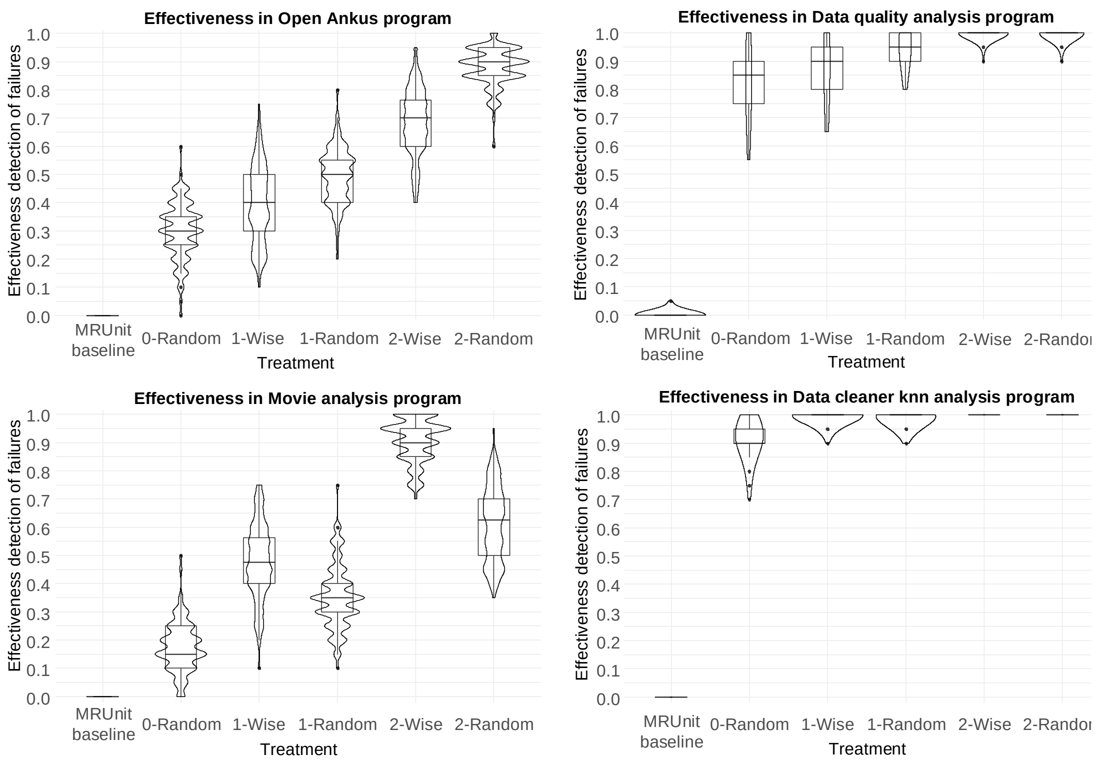

Fig. 15 Distribution of percentage of failures detected per test suite in each program

each test case is reasonable, being in the order of a few milliseconds/seconds. As we explain in detail in the following subsections, the majority of the test cases take less than 1 second to be executed in MRTest, regardless of the approach employed.

During the experiments, MRUnit only detects 4 faults out of 8000, having an effectiveness of 0 in Table 17 due to rounding to two decimal places. These faults are detected because MRUnit sorts the <key, value> pairs when the base configuration is executed and sometimes this change is enough to detect the faults. The execution of the test cases under different configurations can reveal design faults whereas the execution under one configuration could mask them, as occurs in MRUnit. Fig. 15 shows for each program the differences between the tests execution engines in terms of the effectiveness (percentage of failures detected per test suite). This figure uses a violin plot that shows the probability density function and gives a reference with a boxplot. The best testing techniques at detecting failures are 2-Wise and 2-Random, followed by 1-Random and 1-Wise, then 0-Random, and finally MRUnit, which hardly detects any design failures. According to the Wilcoxon Sign Rank Test, all MRtest approaches are significantly better at detecting failures than MRUnit.

In order to compare the best approaches, the Wilcoxon Sign Rank Test is also applied in each program between 2-Wise and 2-Random. Considering the Data Quality Analysis and Data Cleaner Knn Analysis programs, there is no significant difference between 2-Wise and 2-Random ($\text{p-value}_{[1\text{-}10]}$=0.69, $\text{p-value}_{[11\text{-}35]}$=1, $\text{p-value}_{[1\text{-}10]}$=1 and $\text{p-value}_{[11\text{-}35]}$=1, respectively), for the Open Ankus program 2-Random is better ($\text{p-value}_{[1\text{-}10]}$=2.7e-09 and $\text{p-value}_{[11\text{-}35]}$=2.8e-09) and for the Movies Analysis program 2-Wise is better ($\text{p-value}_{[1\text{-}10]}$=3.7e-10 and $\text{p-value}_{[11\text{-}35]}$=3.8e-10).

Fig. 16 shows the aggregation of the data for the 4 programs, 2-Wise being the best approach in detecting failures with a significant difference compared with 2-Random (p-value=0.0043).

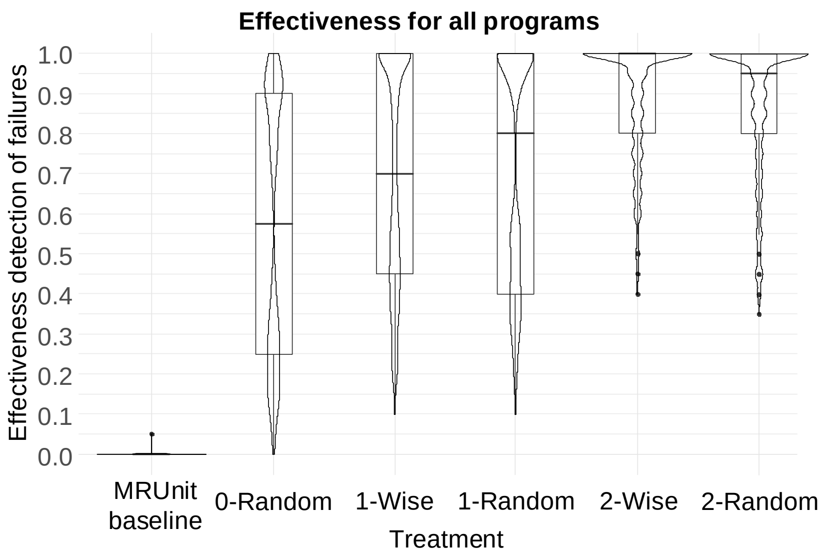

Fig. 16 Distribution of percentage of failures detected per test suite (effectiveness) in all programs

In terms of failure detection effectiveness, all MRTest approaches are better than MRUnit, with 2-Wise and 2-Random standing out, followed by 1-Random and 1-Wise, and finally 0-Random.

III.4.2 Efficiency Experiments

The **goal** of the efficiency experiment is the assessment of how much time is spent during the execution of the test cases. Following the Basili et al. [190] template the goal is: *Analyze* the test case execution engines (MRUnit and MRTest) *for the purpose* of evaluation *with respect to* their efficiency in executing the test cases of the *MapReduce* programs *from the point of view of* the tester and developer *in the context of Big Data* applications. The **planning** of the experiments is described in Subsection III.4.2.1 and their **results** in Subsection III.4.2.2.

III.4.2.1 Efficiency: Setup

In this experiment, 16000 different test cases from 8 real world programs are executed in MRUnit and MRTest in order to analyse the execution time expense per test case. Half of these programs have design faults and their test cases are re-used from the previous experiment, the other 4 programs have no known faults and their functionality is summarized below:

5. Graph clustering [195]: Algorithm to cluster the connected nodes in graphs.
6. Phonetic analysis [193]: Algorithm to clean text data based on the similarities and differences between the phonetic pronunciation.
7. Goldstein analysis [196]: Measure of the conflicts and cooperation between countries based on the Goldstein code.
8. Restaurant analysis [197]: Finds restaurants by cuisine located in safe/unsafe zones in New York.

For each of these programs, 2000 test cases are generated randomly and executed in MRUnit and MRTest with different modes. The **population** is composed by all possible test cases of the *MapReduce* programs, and each one is the **experimentation unit**. All test cases are executed in order to analyse the execution time (**observation**). As in the previous experiment, sets of 20 test cases constitute the test suites that are executed in 6 test engines.

The **dependent variable** or response variable is the execution time of the test case by the different execution engines (**independent variable**). The **baseline** is MRUnit and the **treatments** are MRTest executed in the same way as the previous experiment: {0,1,2}-Random, {1,2}-Wise.

In this experiment, there are other variables that could affect the results and they are treated as **blocking factors** (note that the first two factors were also considered in the previous experiments):

- The size of the input data affects the execution time. The following number of <key, value> pairs are considered during the experiments: between 1 and 10, and between 11 and 35.
- The pseudorandom functionality of the MRTest could introduce noise. To avoid it, the same pseudorandom numbers are used in the MRTest and are generated in a pseudorandom way.
- MRTest executes the test cases with different configurations until a failure is detected or the maximum number of configurations of the approach is reached. Therefore, the execution time can be different whether the program has a fault or not. In this experiment two types of programs are considered: 4 programs with faults reused from the previous experiment and 4 programs without known faults described in this section.
- The execution time could depend on the resources of the computer. All test cases are executed in a commodity computer with a CPU Intel Core i5, 3.20GHz Windows 10 x64, and Java 1.8 with memory generated dynamically up to 250MB.

In order to detail the differences in the execution time, this experiment analyses **descriptive statistics**: a regression model of the execution time in terms of the number of input <key, value> pairs.

As in the previous experiment, the **sampling methods** are consecutive sampling of 8 real world programs and random sampling for the test cases. The number of **trials** per program is again 100 test suites of 20 test cases divided in two sizes of input data: from 1 to 10 <key, value> pairs, and from 11 to 35 <key, value> pairs. Each of these test suites is executed in the MRUnit (**baseline**) and MRTest with different parameters (**treatments**). This type of experiment design is called "within subject design with post-test".

III.4.2.2 Efficiency: Results and Discussion

MRUnit is the most efficient approach because it only executes one *Mapper*, one *Combiner* and one *Reducer*, whereas MRTest executes several of these configurations in order to reveal more faults simulating a production environment. Table 18 summarizes the average execution time of test cases in programs with and without known design faults. MRTest executed in 2-Wise mode is a better approach for detecting failures than Random, but it usually takes longer. In the test cases executed during the experiments, MRUnit takes, on average, a few milliseconds to execute a test case, whereas MRTest usually takes a few milliseconds-seconds, depending on the program and the data that are received. When the program has a fault and MRTest detects it, the execution time is quite similar to MRUnit (x2 or x3) because MRTest finishes after the execution of few configurations. In the case that MRTest does not detect a fault, the execution time on average increases by x200 or x400 from MRUnit, but it remains in the order of milliseconds-seconds per test case.

Table 18 Average execution time of test case through 100 test suites of 20 test cases for each one of the 8 real world programs, in milliseconds

		Programs with known faults				Programs without known faults			
Input size		Open Ankus	Data quality analysis	Movie Analysis	Data cleaner Knn analysis	Graph clustering	Phonetic analysis	Goldstein analysis	Restaurant analysis
Input size	Treatment								
[1-10] <key, value> pairs	MRUnit baseline	51.0	50.9	53.0	54.4	4.9	3.7	3.3	3.8
	0-Random	69.0	69.2	75.1	64.4	193.4	44.1	8.7	7.5
	1-Wise	84.2	94.3	1780.7	80.2	769.9	131.6	21.4	33.3
	1-Random	72.2	72.4	89.2	65.0	501.7	73.2	13.3	11.5
	2-Wise	149.9	145.0	2248.9	70.6	5140.7	384.3	74.8	563.1
	2-Random	77.4	73.5	140.0	64.7	1855.1	195.0	33.9	27.1
[11-35] <key, value> pairs	MRUnit baseline	51.8	50.6	55.3	78.8	7.7	4.8	5.4	5.3
	0-Random	76.0	75.3	93.2	115.0	1183.7	283.7	19.4	16.6
	1-Wise	85.5	104.8	139.4	152.2	3141.6	431.6	29.8	49.9
	1-Random	84.0	78.9	157.7	117.4	3445.0	539.6	36.5	28.6
	2-Wise	128.6	117.3	468.3	123.5	15013.0	1357.9	118.2	2282.4
	2-Random	104.0	78.8	575.7	117.2	13161.2	1606.6	153.5	105.2

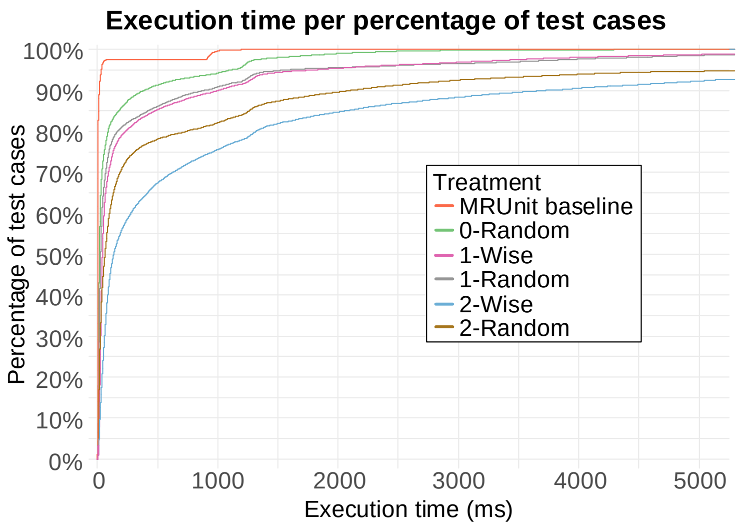

Fig. 17 Accumulated distribution of the test cases execution time

Given a program, there are several test cases that take more time than others, especially when {1,2}-Wise does not cover the test coverage items after trying to generate several configurations. The most expensive test case takes 4.5 minutes for the previous reasons, but in general the test cases are executed in milliseconds or a few seconds, depending on the program functionality and the input received. As Fig. 17 depicts, 75% of the test cases are executed in less than 1 second and 90% in less than 4 seconds.

The execution time depends on several factors, but it increases according to the number of <key, value> pairs in the test case. In Fig. 18 the trend of the execution time based on the number of <key, value> pairs is described for the 4 programs with faults, and in Fig. 19 for the other 4 programs without known faults. This trend in general has more slope in 2-Wise, 1-Wise and 2-Random because these approaches generate and execute more configurations. In these approaches the execution time is more dispersed because it does not only depend on the number of <key, value> pairs, but also on the program and on the data processed. In the case of {1,2}-Wise, the execution time also depends on the non-covered test coverage items, because, for example, in the most expensive test cases, MRTest takes a long time trying to generate values that cover the configurations that cannot be covered. For this reason, the execution time of 2-Wise in the Open Ankus and Data Quality analysis programs decreases according to the input size. When these two programs receive a small amount of input data, the 2-Wise takes time trying to cover the test coverage items. In some cases, 1-Wise is more expensive than 2-Wise because the test coverage items are different. For example, in the Data cleaner Knn analysis program, the second test coverage item of 1-Wise cannot be covered (it requires only one *Reducer* and the program guarantees several) but the approach wastes time trying to cover it.

While MRUnit is not intended to detect these design faults, all approaches of the MRTest are effective enough detecting them, particularly 2-Wise mode. These approaches take a few milliseconds-seconds to execute the test cases and could be a reasonable alternative to detect design failures before they are encountered in production.

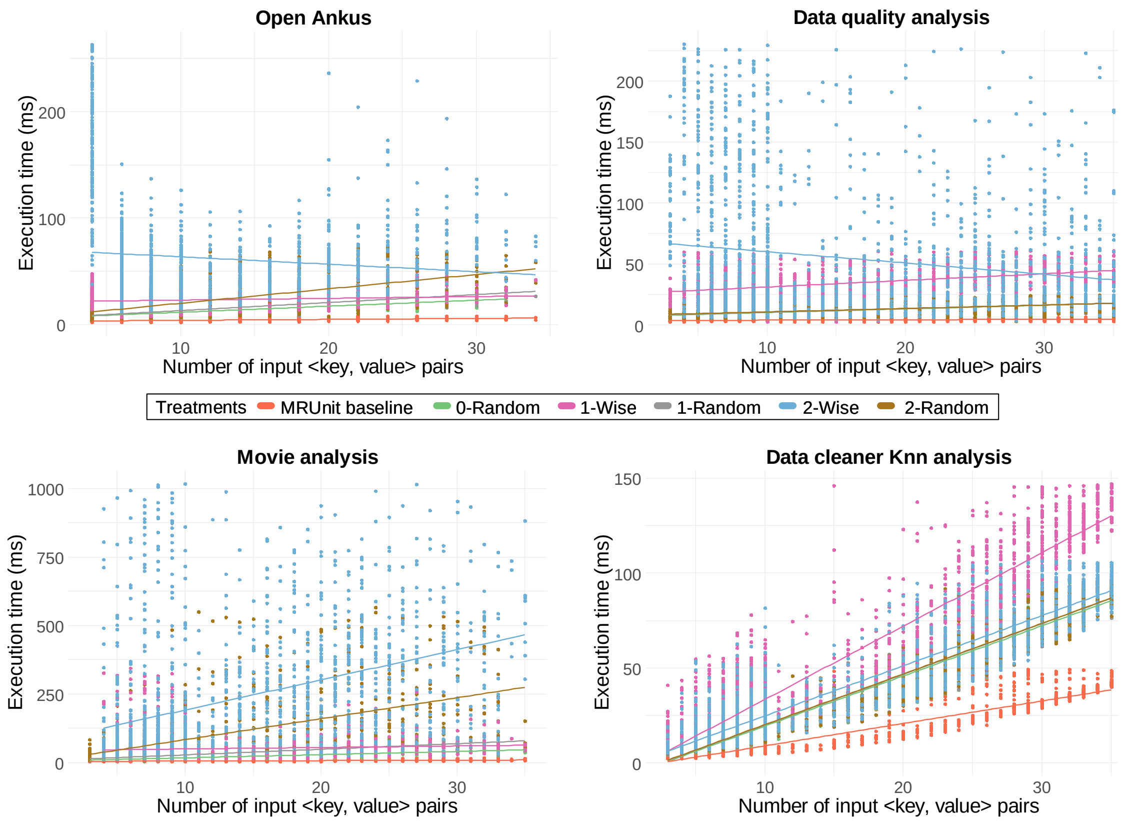

Fig. 18 Execution time of the test cases of programs with known faults according to the number of <key, value> pairs

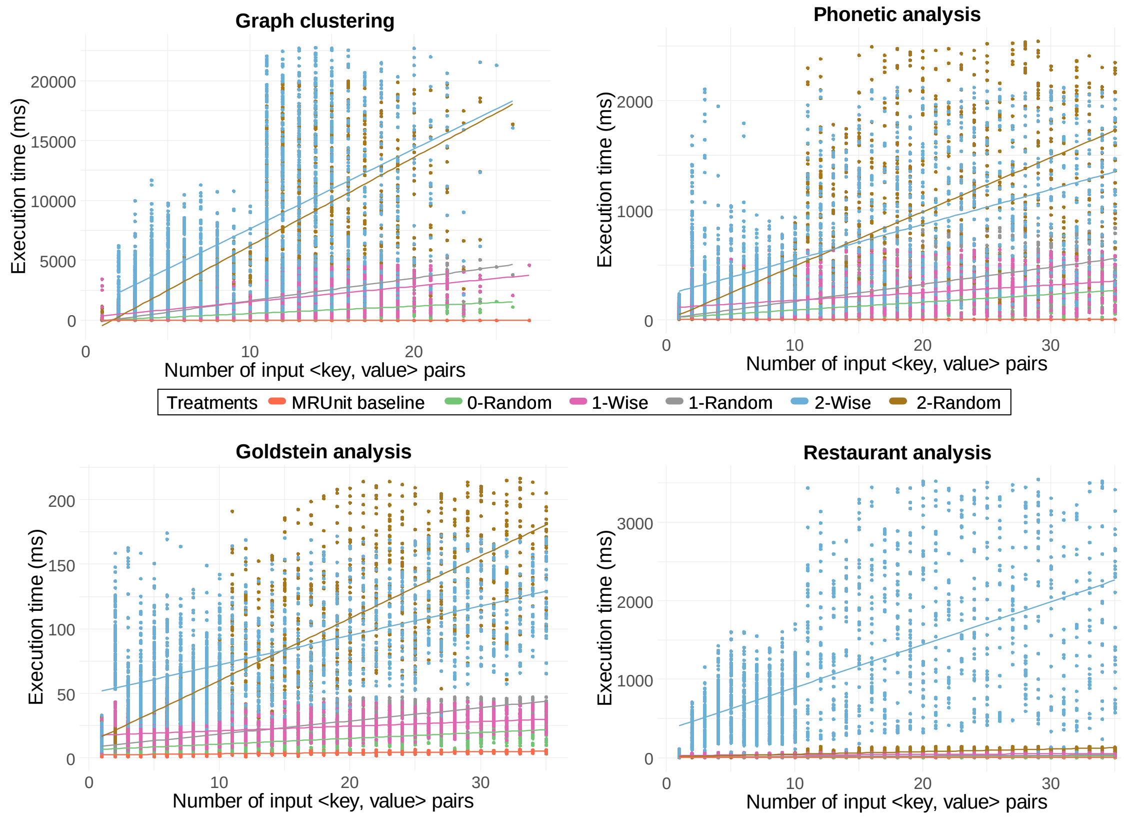

Fig. 19 Execution time of the test cases of programs without known faults according to the number of <key, value> pairs

III.4.3 Discussion of Results

The experiments indicate that both test execution engines proposed in this chapter, MRTest-Random and MRTest-t-Wise, are able to detect within an acceptable time a broad number of failures that are caused by the non-deterministic executions of *MapReduce* programs. Of the two, the MRTest-2-Wise is significantly better at failure detection, and takes an acceptable amount of time to complete the tests as well. In contrast, the MRUnit test execution engine employs less time but it hardly detects any of these types of failures. The remainder of this subsection discusses the limitations of these experiments, including the internal, external and construct threats of validity and their subcategories [194], [198], [199].

The **internal** threats are those issues regarding the causal relationship between independent variables and dependent variables. One part of the experiments analyses the execution time, but some noise can be introduced into the measurements by other operative system tasks (*Confounding effects of variables*). To mitigate this problem, the experiments are executed in the same computer without any other programs operating in the background.

The tool that automates the research, MRTest, can contain faults and other limitations. To mitigate the potential faults of the tool, manual/automatic testing was performed mainly from the functional and performance point of view. This tool may cause side-effects in the programs that perform some communications with external services that are outside the testing context. For example, when the program under test inserts data in an external database, MRTest can perform the insertions for each of the configurations executed. When the external service is fully controllable, then the tester can handle the side-effects inside the test cases.

The **external** threats are those issues that can affect the generalization of the results. The subjects of this experiment are 16000 test cases randomly selected from 8 *MapReduce* programs selected by consecutive sampling. Ideally, the programs should also be selected randomly, but often this is not feasible in software engineering (*Interaction of selection and treatment*). For *Big Data* programs, there is no benchmark of faults and industrial programs are not usually available. This problem is mitigated by using some real-world applications, instead of using programs with seeded faults (hand-seeded faults or mutation faults) that are prone to other external threats [200], [201]. Therefore, there are other issues regarding seeded faults when they are used to evaluate testing techniques. The hand-seeded faults are injected by the expert and they are subjective, decrease the reproducibility of the experiments and are not representative of real faults in terms of easy detection [202]. In contrast, mutation faults are representative of the majority of faults, but this is not the case when the developer implements an incorrect algorithm [203]. The faults pursued by this thesis fall into the previous category of faults that are not possible to substitute with mutations. The faults that are the target of this thesis are caused by incorrect design decisions that lead to the implementation of faulty algorithms, completely different from those of the correct implementation. As such, the injection of mutation faults is not a feasible way to evaluate the testing techniques of this chapter.

The tool that automates the research, MRTest, does not fully support the testing of non-deterministic programs (*Applicability of results across different samples*). This research proposes the execution of the test case in different configurations and finally a metamorphic relationship checks if their outputs are equivalent. The tool only checks if the outputs are equals or not, but this is not enough for non-deterministic programs. To avoid this problem, the tester can implement a function to check if two non-equal outputs are equivalent or not in the non-deterministic program. There are also metamorphic relationships for non-deterministic programs [204], [205].

Other results can be obtained if MRTest generates the configurations in a different way (*Applicability of results when technique is varied*). The configurations are generated based on the combination of different parameters, but there could be more parameters not considered or better ways to generate the configurations such as, for example, using a search-based approach.

The **construct** threats are those issues between the experiment and its underlying theoretical concepts. The test execution engines proposed are only compared against MRUnit despite the fact that there are other ways to automate the testing execution. In general, MRUnit is more standardized and controllable when performing tests in the *MapReduce* applications.

One part of the experiment analyses the efficiency of the test execution engine based only on the execution time measure, but there could be more measures not considered, such as memory (*Mono-operation bias*). To mitigate this problem, the experiments were executed in a commodity computer with few resources. The memory does not appear relevant because its usage was low during the experiments. Furthermore, the tool that automates the research was tested to avoid memory bottlenecks, and some memory leaks of MRUnit were removed.

III.5 Conclusions

The detection of design faults in *MapReduce* depends on the test input data and on the test configurations, i.e. how the test data are executed in parallel. These design faults can be revealed in some executions and masked in others. Thus, although the application may appear to work correctly in the test environment, this might not be the case when it is passed to production because usually these faults are only revealed in aggressive environments. In this chapter, we presented two black-box testing techniques that automatically detect these faults. Given a set of test input data, the testing techniques simulate the execution in infrastructure configurations aimed at revealing the faults, and check that all executions lead to equivalent outputs. These testing techniques are automated in a test execution engine called MRTest.

We performed an empirical study to evaluate the effectiveness and efficiency of the testing techniques proposed (MRTest-Random and MRTest-t-Wise) compared to the XUnit tool of *MapReduce* programs (MRUnit). The results showed that our approaches are more effective in detecting faults while still employing reasonable time. The results also showed that MRTest-t-Wise based on Partition testing detects faults with a significantly lower fraction of tests than MRTest-Random that is based on Random testing.

MRTest enables fine-grained control of the test case execution at the same time as it guarantees its reproducibility in the same circumstances. The simulation of the test case in different production environments can be carried out in a non-intrusive way and with few resources, deploying MRTest on a commodity computer in the laboratory. Furthermore, the testing techniques of this chapter are easy to use because they do not need the expected output to reveal the faults, only the test input data.

IV DEBUGGING

The previous chapter of the thesis proposes a testing technique to detect design faults automatically in the *MapReduce* applications by analyzing the execution of the input data under different infrastructure configurations. Once a design failure is detected, the developer will debug the program to both locate and understand the fault, and fix the program. The design faults of the *MapReduce* are difficult to debug because sometimes are masked and other times are manifested in non-deterministic way. This chapter proposes a framework called MRDebug aimed to help developers during debugging to locate automatically the root cause of the design fault, and isolate automatically the data that trigger the failure.

The root cause of the faults is located through spectrum-based fault localization technique that analyzes the execution of several configurations to extract a pattern of the characteristics that

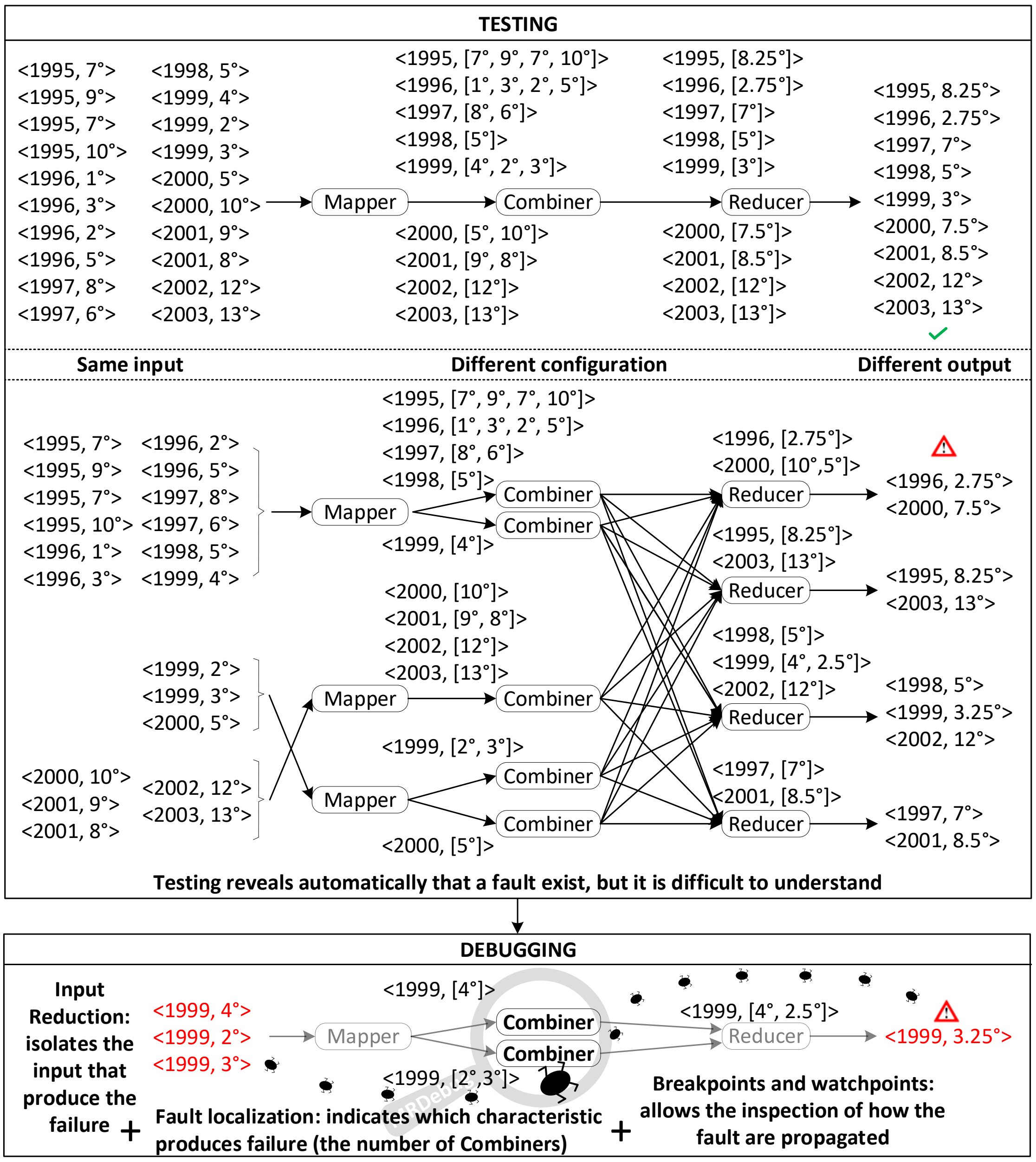

Fig. 20 Automatic testing and debugging

usually triggers the failure and those that usually mask it. This root cause of the fault is an entry point for the developer to understand the fault, but sometimes may not be enough because the test cases process several data with concurrent interactions. To improve the understanding of the fault, MRDebug also isolates/minimizes the data that are relevant to trigger the failure removing these other data that are irrelevant to understand the fault through a search-based algorithm (delta debugging or genetic algorithm).

The majority of this chapter is planned to be published in JCR journal, but the initial work is published in JISBD 2018 [46]. Section IV.1 describes the MRDebug framework: Fault localization (Section IV.2) and input reduction (Section IV.3). The experiments are detailed in Section IV.4. The related work is described in Section IV.5, and finally the Section IV.6 contains the conclusions of the chapter.

IV.1 Debugging Framework: MRDebug

Once the fault was detected through the testing technique of the previous chapter, the debugging phase is started to help developers to understand both the failure and the fault. For example, the fault of the program that calculates the average temperature per year (Section II.1) is depicted in Fig. 20. The first configuration executed does not produces a failure, but the configuration of the middle reveals a fault caused by a wrong design of the application. Despite the test case has only 20 <key, value> pairs, this fault is difficult to both understand and fix. This chapter describes the MRDebug framework that is intended to automatically debug the design faults of the *MapReduce* applications as Fig. 21 summarizes. This framework is able to automatically localize the root cause of the fault (Subsection IV.2), isolate the data trigger the failure (Subsection IV.3), and provide the common debugging utilities.

As a result of debugging, the developer does not obtain the complex configuration that triggers the fault like the middle of Fig. 20, but obtains a more simple configuration like in the bottom of Fig. 20. The input reduction technique automatically minimizes the data that trigger the fault making the wrong distributed processing easier to understand. The localization technique automatically obtains the characteristic of the configuration that trigger the fault. Then the developer can be focus just in one part of the program. Some studies suggest that developers could be beneficed by the integration of debugging techniques [206]. Then to make the debugging more practical, MRDebug also supports the common utilities like breakpoints or watchpoints simulating the distributed execution.

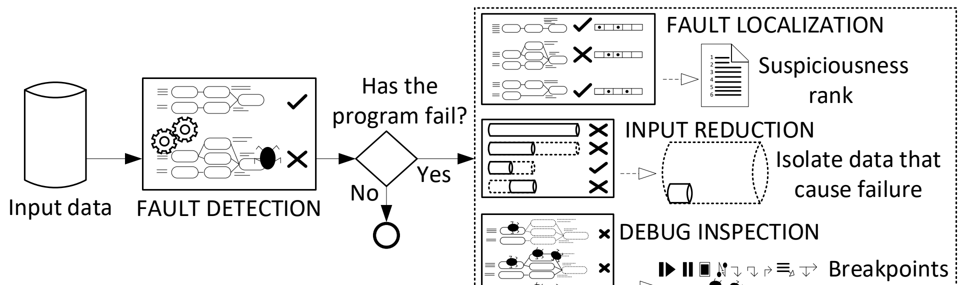

Fig. 21 MapReduce testing and debugging framework

IV.2 FAULT LOCALIZATION

Fault localization techniques aim to locate the root cause of the fault. Among them, the most used in research is the spectrum-based fault localization that analyzes the common characteristics of the test executions that fail and their differences with those that success [207]. These techniques obtain the suspicious cause of the fault following a procedure with the following four parts: (1) definition of the characteristics to be analysed (program spectra), (2) generation of several test cases, (3) execution and monitoring of tests, and (4) analysis of the characteristics executed.

The characteristic analysed are defined according a program spectra [208], [209] such as the coverage of code, parts of the configurations or other resources. The suspicious cause of the fault is obtained analyzing the characteristics covered by not only one test case, but a test suite with different test cases. This analysis yields to different suspicious cause of the fault depending on the quality of the test suite executed, such as the number of test cases [210], the variety of executions [211] and coincidental masking of the faults in executions [212], [213]. The best test cases generated/designed, the more chances to obtain the root cause of the fault.

From each test case executed, the fault localization techniques monitor the characteristics covered to rank the most suspicious cause of the fault. This ranking is obtained analyzing the similarity/distance from the vector that contains the failures of all test cases (failure) to the vector of the coverage of each characteristic in all executions (characteristic covered). Those characteristics that are covered during the failures and also not covered in the succeeded tests, has more chances to be the root cause of the fault. That is, the characteristic that has more suspiciousness to be the root cause of the fault is the one that has a vector of coverage more similar to the failure vector. Then the technique uses a ranking metric to rank each characteristic according to the suspiciousness (similarity). There are several ranking metrics based on binary similarity measures and no one is the best in all domains of fault localization [214].

IV.2.1.1 Fault Localization in MapReduce applications

The fault localization techniques are usually focused on the source code. However, the root cause of the fault is not always the source code, for example the localization techniques of product lines localize the root cause of the faults in features sets instead of the source code [215]. The same happens in the design faults of the *MapReduce* programs because the root cause of the fault can be a characteristic of the distributed execution, such as the number of *Mappers* executed in parallel or the optimizations in the *Combiner*.

A *MapReduce* application can be executed with a configuration that is composed by different characteristics according to its design like the two executions of the top of Fig. 20. The first configuration of the figure masked the failure and has the following characteristics, among others: 1 *Mapper*, 1 *Combiner* and 1 *Reducer*. In contrast, the second configuration has other characteristics that reveal the failure, among others: several *Mappers*, the data is not executed in the same order as in the input, several *Combiners*, and several *Reducers*. The execution of the *MapReduce* programs could success or fail depending on the characteristics of the configuration executed (number of *Mappers*, execution order, distribution of data, and others detailed below).

Fig. 22 summaries the spectrum-based fault localization technique proposed. MRDebug generates and analyzes the characteristics of the different configurations (Program spectra) to obtain automatically the root cause of the fault. Given a configuration that fails, MRDebug generates new configurations changing just one characteristic in order to analyze if this change

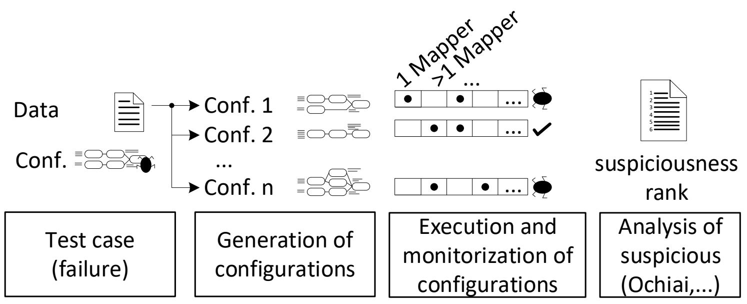
Data
Conf.
Conf. 1
Conf. 2
...
Conf. n
1 Mapper
>1 Mapper
...
suspiciousness rank
Test case (failure)
Generation of configurations
Execution and monitorization of configurations
Analysis of suspicious (Ochiai,...)

Fig. 22 Fault localization technique in MapReduce programs

is able to commute the execution to fail/success (Generation of configurations). These configurations are executed using MRTest as automatic testing technique (Chapter III) collecting the characteristics covered (Execution and monitorization). Then the characteristics of the configurations are analysed together considering the patterns of those that success and those that fail. Finally, the technique obtains a ranking of the characteristics that are more likely to cause the failures (Analysis of suspicious). This technique is detailed below.

Program spectra: The *MapReduce* design faults are caused by characteristics of the distributed executions. Then the program spectra used is focused in these characteristics that were defined in our previous testing technique [41] through input space partitioning: Number of *Mappers* (1 or >1), Data processing order of the inputs (*Mappers* executed in the same order as input or different), Distribution of input data in *Mappers* (equally distributed or not equally), Number of *Combiners* (0, 1, or >1), Distribution of the *Mapper* outputs in *Combiners* (equally distributed or not equally), Data directly from Mapper to *Reducer* (0 or >0), Iterative executions of *Combiner* (1 or >1), and Number of *Reducers* (1 or >1). In total, these 17 characteristics can be combined in different configurations. In each execution, the technique collects the characteristics covered to obtain the one that usually triggers the failure.

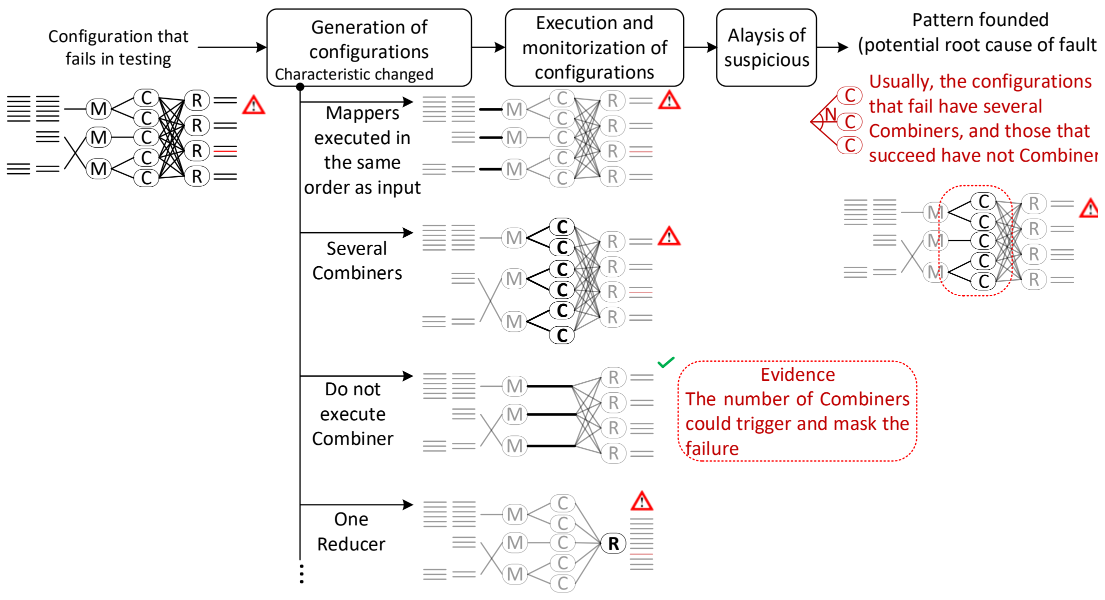
Configuration that fails in testing
Generation of configurations
Characteristic changed
Execution and monitorization of configurations
Alaysis of suspicious
Pattern founded (potential root cause of fault
Usually, the configurations that fail have several Combiners, and those that succeed have not Combiner
Mappers executed in the same order as input
Several Combiners
Do not execute Combiner
One Reducer
Evidence
The number of Combiners could trigger and mask the failure
M
C
R
N

Fig. 23 Generation of configurations during fault localization

Generation of configurations: MRDebug starts with one execution/configuration and generates more configurations with different characteristics aimed to provide enough information to locate the root cause of the fault. The configurations are generated based on the Lewis counterfactual theory of causality [216] like other fault localization techniques [217]. Then, MRDebug modifies the configuration that failed in testing generating several new configurations with just one characteristic change in order to analyse if this change is able to mask the fault or continue with the failure. Fig. 23 summarizes the generation of these new configurations that changes only one characteristic: varying the execution order of the *Mappers*, executing the configuration with several *Combiners*, without *Combiner*, with one *Reducer*, and so on with one characteristic changed to cover new characteristics. These new configurations that are close to the configuration that fails aims to avoid the coincidental masking of the faults.

Execution and monitorization: The configurations generated in the previous step are executed and the design failures triggered are collected. During the execution, MRDebug also tracks the characteristics covered by each configuration.

Analysis of suspicious: Once the configurations are generated and executed, the characteristics of these configurations are analysed to obtain those that are more prone to be covered when the execution fails (Characteristic covered = TRUE, failure = TRUE) and those that usually are not covered when the execution success (Characteristic covered = FALSE, failure = FALSE). MRDebug supports the most common 52 ranking metrics [188], [207], [218].

Example: Fig. 23 depicts all procedures of the fault localization in the program that calculates the average temperature per year starting with the configuration that causes the failure in the middle of Fig. 20. This program fails because the *Combiner* calculates the average temperature with the temperatures available locally, but *Reducer* is not able to obtain the global temperature with the local temperatures. The configuration that triggers the failure is composed by the following characteristics, among others: *Mappers* that are executed in different order than the input data, some *Mappers* have one *Combiner*, other *Mappers* have several *Combiners*, and one *Reducer*. Per each characteristic of this configuration, 5 new configurations are generated varying (1) the order of the *Mapper*, (2) with several *Combiners*, (3) without *Combiner*, (4) with one *Reducer*, and so on (Generation of configurations). Note that Fig. 23 only depicts 1 of the 5 configurations generated. Next, each configuration is executed to check if the characteristic changed is able to mask the failure or still trigger it (Execution and monitorization). For example, the first configuration changes the execution order of *Mapper* and obtains the same output (failure regardless of the execution order), then the order of *Mappers* apparently does not produce the failure. However, the third configuration changes the number of *Combiner* to 0 and the output of the test cases change from failure (one-several *Combiners*) to success (zero *Combiners*). This is an evidence that the number of *Combiners* is suspicious to cause failures: the failure is triggered with several *Combiners* and masked with zero *Combiners*. This last analysis is done through the ranking metric M2 obtaining that the most suspicious characteristic is the execution of several *Combiners* (Analysis of suspicious). Effectively, this *MapReduce* program was wrongly design because does not admit this *Combiner* functionality and the failures are triggered when several *Combiners* are executed. This information allows the developer to fix the program removing the *Combiner* (patching the failure instead to correct the fault [206]) or creating a new design of the program that supports other kind of *Combiner*.

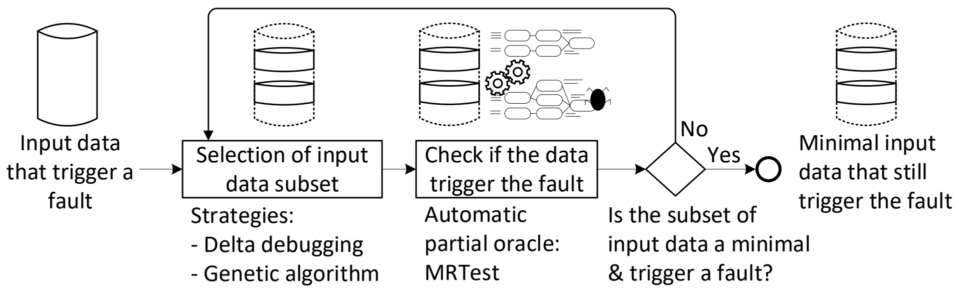
Input data that trigger a fault
Selection of input data subset
Strategies:
- Delta debugging
- Genetic algorithm
Check if the data trigger the fault
Automatic partial oracle: MRTest
No
Yes
Is the subset of input data a minimal & trigger a fault?
Minimal input data that still trigger the fault

Fig. 24 Input reduction technique

IV.3 Input reduction

The fault localization technique obtains automatically a ranking of the most suspicious root causes of the fault. This ranking is an entry point for the developer to continue with the debugging to both understand and fix the fault. Usually the root cause of the fault is not enough to understand the fault, and the developers need more contextual information [206]. In the example program of the previous subsection, the fault localization obtains that the most suspicious cause of the fault is the execution of several *Combiners*, but this is not enough to understand completely the fault. Even without very large test input data as in the middle of Fig. 20 (20 <key, value> pairs), the fault is not easy to understand because only 3 <key, value> pairs trigger the failure and the other data difficult the understanding. Highlight those data related to the fault can improve the understanding of the faults as other researchers suggest [206].

The MRDebug framework proposes to highlight the relevant test data minimizing/isolating the data that trigger the failure to improve the understanding of the *MapReduce* design faults. As Fig. 24 summarizes, this input reduction approach generates iteratively new subsets of the test input data that still trigger the failure removing/unselecting those irrelevant <key, value> pairs. Each one of these subsets are used to execute the program and determine if the data still trigger the failure or not. In each iteration, the new subsets of data are generated by means of search-based strategy [219]: delta debugging that is a greedy approach focused on local search [48] (Subsection IV.3.1.2), or genetic algorithm (Subsection IV.3.1.1) [50], [51]. These subsets are executed using MRTest as automatic testing technique/oracle (Chapter III) that determinises if the subset of data is able to trigger the failure (feasible solution) or not (unfeasible solution). After several iterations/optimizations, the technique obtains automatically a minimal/small subset of data that still trigger the fault. Finally, these small data make the fault more easy to understand. For example, the bottom of Fig. 20 depicts a simple test case with only 3 <key, value> pairs that is more easy to analyse than the test case of the middle. In the following subsections, the genetic algorithm and delta debugging strategies are detailed.

IV.3.1.1 Genetic Algorithm

Genetic algorithms solve problems based on the nature selection of the Darwin theory about evolution. The solutions of the problem are represented by one individual and codified in the chromosome. The goal of these algorithms is to generate better individuals that inherit the best genes through reproduction and mutation as occurs in nature.

The genetic algorithm proposed to reduce/isolate the data of the test case (individual) that trigger the failure is summarized in Fig. 25. At first point, the technique generates and executes several subsets of data randomly (initial population). Then the algorithm evaluates how good

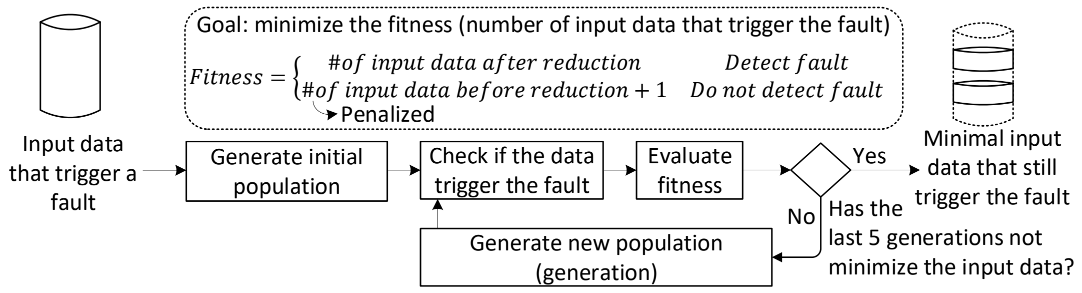
Goal: minimize the fitness (number of input data that trigger the fault)
$Fitness = \begin{cases} \#of\ input\ data\ after\ reduction & Detect\ fault \\ \#of\ input\ data\ before\ reduction + 1 & Do\ not\ detect\ fault \end{cases}$
Penalized
Input data that trigger a fault
Generate initial population
Check if the data trigger the fault
Evaluate fitness
Yes
Minimal input data that still trigger the fault
No
Has the last 5 generations not minimize the input data?
Generate new population (generation)

Fig. 25 Genetic algorithm to reduce the input data

each subset is (fitness function), considering that the subset is as good as less data that still trigger the failure has. Subsequently, the algorithm improves the actual subsets of data trying to reduce their number of data (generation of new population). This algorithm also generates iteratively new subsets of data until there is no improvement in the last generations (fitness equally in last generations). As result, the algorithm obtains a minimal data or at least an enough small data that still trigger the failure.

Individual (chromosome): The solution of this problem, also called individual, is one of the possible subsets of the test input data (<key, value> pairs). Given an input data that failed during testing with N <key, value> pairs, the search space is composed by (2^N)-1 possible subsets of data (individuals). The subset of data of each individual is encoded in a chromosome composed by a binary string schema [50] as in the Fig. 26. The genes of the chromosome indicate if the <key, value> pair is selected (1) or not (0). Then this chromosome can be decoded in a subset of data that could still trigger the failure or success. The best individuals are those that trigger the failure with less data.

Initial population: The initial population is composed by several individuals randomly generated together with another individual composed by all input data that failed in testing. This last individual is used to start the reduction approach with the worst feasible solution.

Some researchers suggest an initial population with large number of individuals [220], but similar results can be achieved with smaller number initial population after several generations [221]. The use of small number of individuals like 20 is also preferable when the cost of each

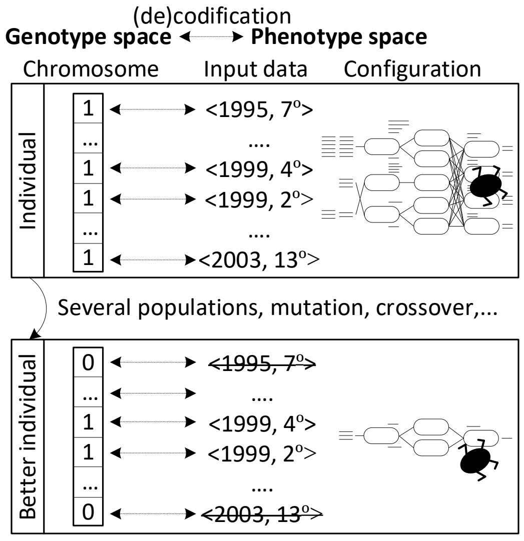

Fig. 26 Input reduction technique

generation/individual can be high, as in our case. As the experimentation of Section IV.4 details, sometimes the execution can take several seconds/minutes because the individuals are executed by MRTest (Chapter III).

Execution of individuals: Each individual composed by a subset of input data is executed by MRTest [41]. Then the algorithm obtains automatically if the individual has data able to trigger the failure (feasible individual) or not (infeasible individual).

Fitness: Each individual is evaluated by a fitness function that indicates how good is the subset selected. The best ones are those that triggers the failure with few data, whereas the worsts are those that either cannot trigger the failure or have lots of data.

The goal is to minimize the fitness considering the feasible and infeasible individuals. The feasible individuals are those that triggers the failure, and their fitness is defined as the number of the data selected. In contrast, the infeasible individuals are those that are not able to trigger the failure. This infeasible individuals are penalized by a maximum fitness value (death penalty [222]). One good practice to stablish this penalty is by the *minimal penalty rule* that consist in the lower penalty that make the individual still worse than the worst feasible individual [223]. In the input reduction problem, the worst feasible individual always contains all input data (fitness = number of input data that failed during the testing). Then the fitness is defined as: (1) ”the number of input data that failed in testing + 1” for the individuals that are not able to trigger the failure, and (2) “the number of data selected” for those individuals that trigger the failure.

Generation of new population: The initial population is iteratively improved with new individuals that inherits the best chromosome genes, these are those <key, value> pairs that are relevant to trigger the failure. This new generations of individuals are created using all of the following operations: (1) elitism, (2) asexual reproduction, (3) sexual reproduction, (4) mutation, (5) non-duplicated individuals, and (6) generational replacement. These operations aims to obtain both the maximum reduction of subset of input data that trigger the failure (local minimal), and diversity of these data (skip local minimal to reach global minimal).

From one generation to the next, the best individual is preserved (elitism [224]) to provide two advantages: guarantees that each generation has at least one subset of data that triggers the failure (feasible solution), and the new generation has at least one individual that has either the same or less number of <key, value > pairs than the previous generation. This best individual can be better if is able to still trigger the failure with at least one <key, value> pair less. Then the algorithm also explores the close subsets of this elite individual generating new similar subsets reducing just one <key, value> pair. This operation is called asexual reproduction of one individual mutating one random gen of the chromosome [225] to reduce one <key, value> pair. The previous two operations (elitism and asexual reproduction) search only a small part of the search space. Then these individuals generated from the asexual reproduction trend to improve the solutions towards a local minimal whereas the other operations trends to skip the local minimal.

The algorithm also provides diversity in the subsets of the test input data selected to search the space more broadly and skip local minimal aimed to reach the global minimal. The remainder individuals of the new generation are generated through sexual reproduction of previous generation and little mutations. All individuals of the next generations are generated mixing two parents selected through the roulette wheel method [51]. This method selects the individuals randomly but weighted according to the fitness value, then the individuals with less data that triggers the failure have more chances than those that either not trigger the failure or have more data. For each two individual parents selected, two individual offspring are generated using one-point crossover [51] to join different parts of the two subsets aimed to still trigger the failure but with more diversity of data. This method generates one individual with the first part of the input data from the father chromosome and the remainder part of data from the mother, and also another individual with the opposite (the first part is from mother and the second from father). The point that separates/crosses the chromosome genes from the father to the mother is selected randomly. Then each offspring is a mix of the father subset and mother subset (previous generation). Finally, each gen of the chromosomes can be mutated swapping the value to introduce littles changes selecting or unselecting one <key, value> pair. Some researchers suggest to perform the mutations with low probability [224], for example 1%.

In this algorithm, each new individual generated must be a completely new individual (not duplications [226]). In case that one new individual was generated identically before, this individual is replaced by a new random subset of the test input data. The completely new individuals generated in each generation replace the previous generations (generational replacement [51], [227]).

Termination criteria: New generations are created until there is no improvement in the best individual fitness during the last generations (K-iterations [228]).

Example: Fig. 26 depicts part of the input reduction technique using the input data that fail in the top of Fig. 20. An initial population is generated with one individual that contains all data that trigger the failure (top of Fig. 26) and other 19 random subsets of these data (not represented in Fig. 26). Then MRDebug uses the genetic operators to iteratively generate new better individuals that modify the subsets of the test input data and reduce their number of <key, value> pairs. Finally, the algorithm converges after 7 generations to the minimal subset of data that still trigger the failure as in the bottom of Fig. 26. This subset of the test input data is also represented in the bottom of Fig. 20 containing only 3 <key, value> pairs after MRDebug removed the irrelevant data to enhance the understanding of the fault.

IV.3.1.2 Delta Debugging

Delta debugging is an approach proposed by Zeller et al. to simplify the test cases isolating the inputs [48]. This isolation/minimization is done through a greedy search-based algorithm aimed to find the first local minimal using a binary-search strategy recursively. At first point, the algorithm proposed by Zeller et al. divides the search space in two halves (binary-search). If one of the halves still contains the solution, then is again divided deeper and the other halve is discarded (greedy). If none of the halves contains the solution, then the granularity of the search space is increased dividing again one of the halves in two (binary-search). This approach is applied recursively until reach the first local minimal.

The details of the technique are in the original paper of Zeller et al. [48], [49]. MRDebug only adapts delta debugging to the input reduction problem of the *MapReduce* applications using MRTest (Chapter III) as automated testing technique/oracle. The search space is composed by all input data that trigger the failure, then the data are iteratively divided into two subsets (binary-search). If one of these subsets still trigger the failure, then this subset is divided again into two subsets until reach a local minimal, and the other subset is discarded (greedy). In the other case that none of the two subsets of data are able to trigger the failure, the granularity is increased dividing one of the subsets into two sub-subsets that are combined with the other subset generating a new search space. The approach is applied recursively until reach a local minimal subset of data that still trigger the failure.

IV.4 Experiments

The *goal* of the experiments is to evaluate both the effectiveness and efficiency of the debugging techniques proposed in this chapter to understand the *MapReduce* design faults: fault localization technique (Subsection IV.4.1) and input reduction technique (Subsection IV.4.2). Finally, Subsection IV.4.3 discuss the results and the limitations.

IV.4.1 Fault Localization experiments

The *goal* of this experiment is the assessment of how effective and efficient the fault localization technique is locating the root cause of the *MapReduce* design faults. The research questions are:

RQ7. Is the fault localization technique better providing the root cause of the *MapReduce* design faults than a random location (baseline)?

RQ8. The fault localization technique obtains a ranking of the *MapReduce* characteristics that could be the potential root causes of the design fault. How many characteristics should be analyzed until reach the root cause of *MapReduce* design fault?

RQ9. Which ranking metrics of those used in fault localization are better to rank the root causes of the *MapReduce* design faults?

RQ10. How much execution time is employed by the fault localization of the *MapReduce* design faults?

RQ11. The localization technique analyzes several configurations previously generated. How many *MapReduce* configurations must be generated by the fault localization technique to achieve a good balance between maximum rate of design faults located and low execution time?

The setup of the experiments aimed to answer the previous research questions is described in Subsection IV.4.1.1. The results are detailed in Subsection IV.4.1.2.

IV.4.1.1 Fault Localization experiment: setup

In this experiment, 8000 different test cases from 4 real-world programs (2000 test cases per program) are analysed by the fault localization technique to evaluate their capability to localize the root cause of the faults. These 4 programs are the same used in the previous chapter and each one has a known design fault: (1) *Open Ankus* [170] is a recommendation system that fails when the recommendation data about the same item are split and parallelized, (2) *Data quality analysis* [191] measures the quality of the data interchanged by companies and fails when the data are not processed in the same order as in the input due a parallelization issues (the fault is actually fixed), (3) *Movie analysis* [192] obtains statistics about movies and fails due a wrong implementation of *Combiner*, and (4) *Data cleaner Knn analysis* [193] is a machine learning program to clean data that fails when there are several *Mappers* each one trying to access to non local available data.

The *population* of the experiment is composed of all test cases that trigger *MapReduce* design faults. Each of these test cases is then taken as the *experimentation unit*, and the *observations* are: the ranking of the potential root causes of faults (suspiciousness rank) obtained by the fault localization technique (RQ7-RQ9 and RQ11), and execution time (RQ10). The *dependent variable* or *response variable* is: position of the root cause of fault in the suspiciousness rank (RQ7-RQ9 and RQ11), and execution time in milliseconds (RQ10). The *baseline* is the random localization that obtains a random suspiciousness rank, and the *treatments* are the fault localization technique using the most common 52 ranking metrics [188], [207], [218]: Ample, Ample2, Anderberg, Arithmetic Mean, Baroni Urbani and Buser, Binary, Braun Banquet, Cbi Increase, Cbi Log, Cbi Sqrt, Cohen, Dennis, Dice, Euclid, Fleiss, Fossum, Goodman, Gower, Hamann, Hamming, Harmonic Mean, Jaccard, Kulczynski, Kulczynski2, M1, M2, Mccon, Michael, Minus, Mountford, O, O^P, Ochiai1, Ochiai2, Overlap, Pearson, Phi Geometrical Mean, Rogers and Tanimoto, Rogot1, Rogot2, Rusell And Rao, Scott, Simple Matching, Sokal, Sorensen Dice, Tarantula, Tarwid, Wong1, Wong2, Wong3, Wong3 Prime, and Zoltar.

These experiments could be affected by the input data size, then a *blocking factor* is stablished with two different sizes of data as in the previous experiments [41]: small size (between 1 and 10 <key, value> pairs) and a larger size for functional testing purposes (between 11 and 35 <key, value> pairs).

In the experiments, two *sampling methods* are used: consecutive sampling to select the *MapReduce* programs and random sampling to select the test cases. Ideally, the subject programs should be selected randomly, but as in the case in many software engineering experiments, this is not viable [194]. As such, four real-world programs that contain a known fault are selected instead.

These experiments answer the research questions using different statistical measures. Whereas RQ10 is answered analysing the execution time trend because is focused on efficiency, the other research questions analyse the position of the root cause of the fault inside the suspiciousness rank. RQ7 is answered comparing the positions of the root cause of the fault that are provided by both random localization and fault localization. The comparison is done by the non-parametric *statistic test* Wilcoxon Sign Rank test that measures the differences among the paired medians with the following one-tail null hypothesis: H_{01}: median(rank position of the root cause of the fault by fault localization) = median(rank position of the root cause of the fault by random localization). This kind of evaluation is also done in other fault localization studies [229].

RQ8 is answered using one of the most used evaluation metrics in fault localization [188], the EXAM score [230]: percentage of the suspiciousness rank that must be analysed until reach the position of the root cause of the fault.

Another metric to evaluate the fault localization techniques is the AUC (area under curve) that allows the comparison among ranking metrics [231]. AUC consider the position of the root cause of the fault in each test case. AUC is defined as the sum per each test case of the percentage of suspiciousness rank not analyzed. RQ9 is answered using AUC normalized between 0 and 1.

The RQ11 is focused on how much configurations should be generated and analyzed to obtain good rate between effectiveness and efficiency. Intuitively, more number of configurations yields on better results, but also on more execution time. In other domains, Abreu et al. [210] study empirically the relation between the fault localization accuracy and the number of analysis performed. In our domain these number of analysis (number of configurations generated) could vary and then are also empirically studied. The research question is answered though the execution of all experiments varying the number of configurations and analyzing the trend of the AUC normalized along the different ranking metrics.

IV.4.1.2 Fault Localization experiment: results

The localization technique generates and analyzes several configurations to obtain the root cause of the faults. Depending on the number of configurations generated, the results varies a

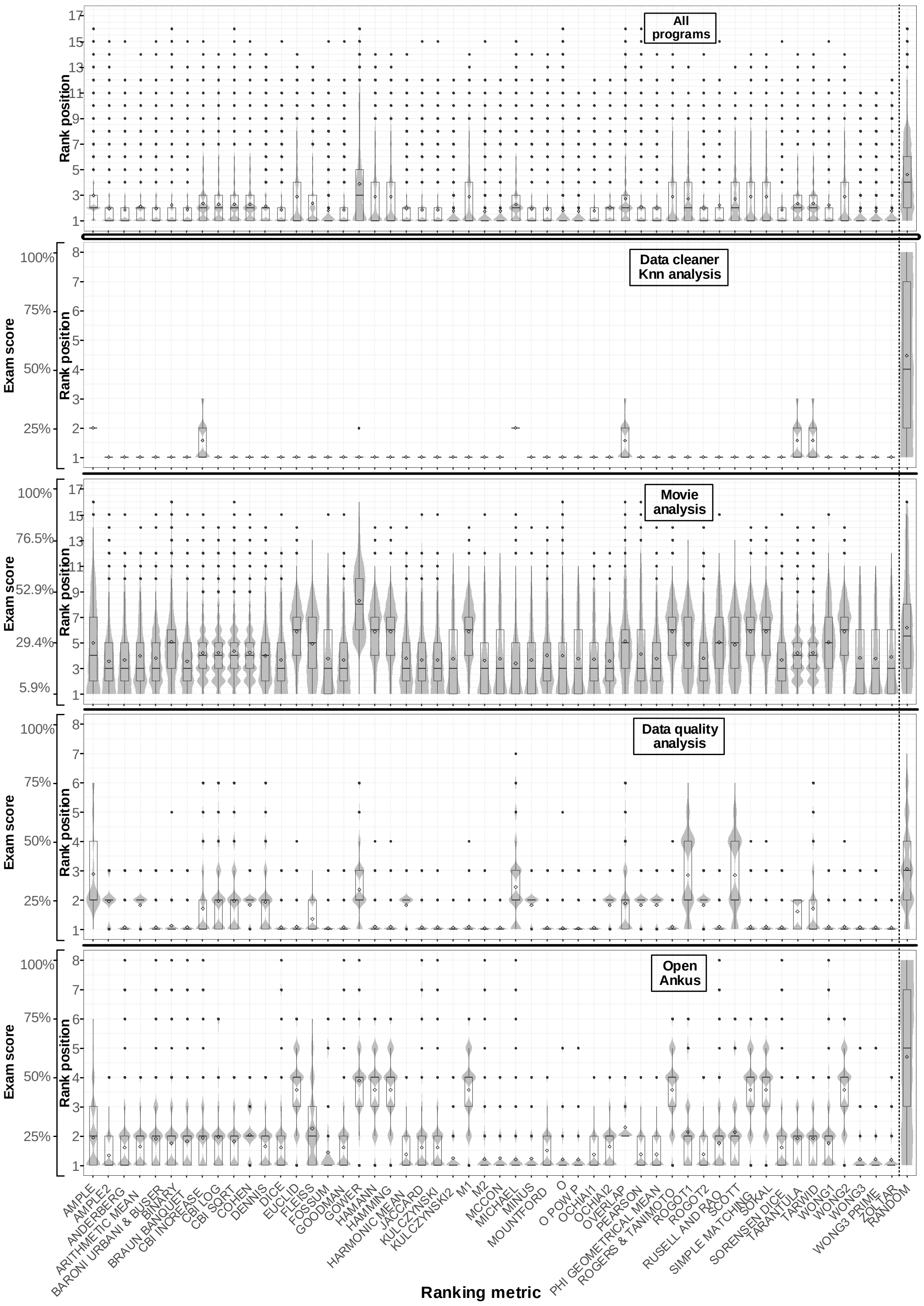

Fig. 27 Distribution of Exam score (% of characteristics examined in the ranking to detect the root cause of fault) and ranking position of the root cause according to teach ranking metric in each program

bit. As we can see below in the answer of RQ11, the results are enough good generating and analyzing just only one configuration per each characteristics of configurations. These results are improved when more configurations are generated and analyzed. The best number of configurations in terms of efficiency and effectiveness is 5 according of RQ11. Then MRDebug use by default 5 as number of configurations per each characteristic, and the remainder experiments for RQ7-RQ10 evaluates the localization technique with this default option.

Fig. 27 details the distribution of the position of the root cause of the fault in the suspiciousness rank obtained automatically by the fault localization technique. In the rigth of the figure is also detailed the baseline (random localization). Table 19 summarizes the p-values that compares the localization technique and the random localization. Regardless of the ranking metric, the fault localization technique is better than random localization with a p-value lower than 0.05. There are two exceptions in *Data quality analysis* program, Rogot1 and Scott, with 11-35 <key, value> pairs as input. In the *movie analysis* program there are also 9 ranking metrics that are not significantly better than random localization, but the other 43 ranking metrics achieve significant better results than random localization. In the aggregation of the 4 programs, all ranking metrics are significantly better than random and the null hypothesis H_{01} is rejected according to Wilcoxon Sign Rank test. Then **the answer of RQ7 is that the localization technique achieves in general significant better results than random localization**.

According to Fig. 27, the fault localization technique usually ranks the root cause of the fault in the first top positions. There are some exceptions in the *movie analysis* program because its root cause of the fault is complex and then difficult to localize. Despite the program does not support the *Combiner*, not always triggers the fault when a *Combiner* is executed. The fault is triggered/masked depending on several factors such as if *Combiner* is executed or not, the logic of the program, and the distribution of the data in the configuration. These issues add noise and the technique sometimes ranks in the first positions characteristics that, although are not the root cause of fault, are very related. Then the root cause of the fault of the *movie analysis* program are usually ranked in in the firsts four positions, that is in less than the 23.5% top of the rank (Exam score). However, the fault localization still provides better results than random. In the remainder of programs, and in the aggregation of all programs, the root causes of the fault are in the top positions of the rank. Then **the answer of RQ8 is that the root causes of fault are in general ranked in the first positions**.

Despite all ranking metrics of the literature, there is no one formula that outperforms the rest in all contexts [214]. In our domain, all ranking metrics not only are in general significantly better than random localization, but also achieve good results. Table 20 summarizes and sorts ranking metrics according to the AUC normalized. After all experiments, **the answer of RQ9 is that the best ranking metrics to locate *MapReduce* design faults are M2, O^P and Wong3 prime**.

From the previous ranking metrics, Fig. 28 describes the accumulated percentage of faults localized at each ranking position. The root causes of fault are ranked in the first position in more than 85% of times for *Open Ankus*, more than 94% for *Data quality analysis*, more than 25% for *movie analysis*, and 100% for *Data cleaner knn analysis*. When the data of all programs are aggregated, in more than 75% of times the root cause of the fault is ranked in the first position, and more than 85% in the second position. Then more information is added in **the answer of RQ8: the root cause of the fault is in general ranked in the first positions (more than 75% of times in the first position and more than 85% of times in the second position)**.

Table 19 P-values of the experiments performed in each program according to the ranking metrics used in fault localization

	Programs								
	Open Ankus		Data quality analysis		Movies analysis		Data cleaner Knn analysis		All programs
Ranking metric	[1-10]	[11-35]	[1-10]	[11-35]	[1-10]	[11-35]	[1-10]	[11-35]	
AMPLE	6.3E-127	1.6E-117	0.01117	2.3E-05	7.0E-24	0.0002	2.2E-138	2.2E-138	~ 0
AMPLE2	7.9E-145	1.6E-139	1.0E-88	4.6E-70	1.9E-60	2.3E-57	1.4E-166	1.4E-166	~ 0
ANDERBERG	6.6E-140	3.9E-131	1.8E-142	2.8E-147	4.4E-53	3.3E-55	1.4E-166	1.4E-166	~ 0
ARITHMETIC MEAN	3.1E-138	2.5E-132	2.4E-105	1.7E-86	2.0E-39	1.0E-46	1.4E-166	1.4E-166	~ 0
BARONI URBANI AND BUSER	1.2E-131	5.1E-122	1.7E-142	2.8E-147	1.1E-55	4.0E-51	1.4E-166	1.4E-166	~ 0
BINARY	5.9E-139	3.2E-129	2.1E-139	1.1E-147	4.1E-10	1.0E-15	1.4E-166	1.4E-166	~ 0
BRAUN BANQUET	5.1E-134	9.6E-125	2.4E-142	2.8E-147	1.9E-64	1.2E-60	1.4E-166	1.4E-166	~ 0
CBI INCREASE	2.4E-129	7.2E-123	1.3E-115	1.3E-68	7.1E-36	3.0E-40	2.8E-166	2.8E-166	~ 0
CBI LOG	2.5E-126	3.2E-105	6.0E-90	5.0E-55	3.4E-35	4.2E-37	1.4E-166	1.4E-166	~ 0
CBI SQRT	3.5E-133	6.5E-129	6.4E-89	4.4E-55	8.7E-30	1.7E-33	1.4E-166	1.4E-166	~ 0
COHEN	1.1E-125	3.9E-121	1.5E-104	1.7E-86	6.7E-36	3.4E-37	1.4E-166	1.4E-166	~ 0
DENNIS	3.2E-138	4.6E-132	1.4E-96	1.1E-56	1.0E-44	3.0E-44	1.4E-166	1.4E-166	~ 0
DICE	6.6E-140	3.9E-131	1.8E-142	2.8E-147	4.4E-53	3.3E-55	1.4E-166	1.4E-166	~ 0
EUCLID	3.0E-52	4.4E-24	1.3E-140	2.8E-147	0.1687	0.2596	1.4E-166	1.4E-166	~ 0
FLEISS	2.7E-118	2.2E-86	5.8E-125	4.1E-133	4.6E-24	5.0E-07	1.4E-166	1.4E-166	~ 0
FOSSUM	6.7E-141	1.7E-122	4.0E-146	1.8E-148	3.0E-43	2.9E-50	1.4E-166	1.4E-166	~ 0
GOODMAN	6.6E-140	3.9E-131	1.8E-142	2.8E-147	4.4E-53	3.3E-55	1.4E-166	1.4E-166	~ 0
GOWER	1.9E-33	4.3E-16	4.3E-28	3.9E-46	1	1	1.4E-166	1.4E-166	1.27E-110
HAMANN	3.0E-52	4.4E-24	1.3E-140	2.8E-147	0.1687	0.2596	1.4E-166	1.4E-166	~ 0
HAMMING	3.0E-52	4.4E-24	1.3E-140	2.8E-147	0.1687	0.2596	1.4E-166	1.4E-166	~ 0
HARMONIC MEAN	2.4E-144	3.2E-139	2.4E-105	1.7E-86	5.1E-49	5.4E-51	1.4E-166	1.4E-166	~ 0
JACCARD	6.6E-140	3.9E-131	1.8E-142	2.8E-147	4.4E-53	3.3E-55	1.4E-166	1.4E-166	~ 0
KULCZYNSKI	6.6E-140	3.9E-131	1.8E-142	2.8E-147	4.4E-53	3.3E-55	1.4E-166	1.4E-166	~ 0
KULCZYNSKI2	1.5E-146	1.2E-141	1.0E-144	7.3E-148	6.4E-41	2.8E-53	1.4E-166	1.4E-166	~ 0
M1	3.0E-52	4.4E-24	1.3E-140	2.8E-147	0.1687	0.2596	1.4E-166	1.4E-166	~ 0
M2	9.3E-147	8.6E-141	4.0E-146	1.8E-148	9.3E-47	2.5E-58	1.4E-166	1.4E-166	~ 0
MCCON	1.5E-146	1.2E-141	1.0E-144	7.3E-148	6.4E-41	2.8E-53	1.4E-166	1.4E-166	~ 0
MICHAEL	5.8E-147	8.8E-141	1.1E-62	3.2E-21	6.0E-67	2.3E-63	2.2E-138	2.2E-138	~ 0
MINUS	3.7E-147	1.1E-141	2.4E-105	1.7E-86	7.0E-47	3.6E-60	1.4E-166	1.4E-166	~ 0
MOUNTFORD	2.2E-141	2.6E-135	2.2E-144	7.3E-148	4.8E-37	8.3E-44	1.4E-166	1.4E-166	~ 0
O	5.9E-147	8.7E-142	3.2E-146	2.6E-148	3.0E-31	8.4E-52	1.4E-166	1.4E-166	~ 0
O POW P	4.0E-147	8.1E-142	7.9E-147	1.8E-148	9.9E-39	7.6E-54	1.4E-166	1.4E-166	~ 0
OCHIAI1	7.6E-145	5.6E-139	1.1E-144	7.3E-148	1.3E-46	3.1E-54	1.4E-166	1.4E-166	~ 0
OCHIAI2	2.2E-139	3.7E-132	2.4E-105	1.7E-86	5.9E-68	9.7E-56	1.4E-166	1.4E-166	~ 0
OVERLAP	6.1E-116	1.6E-105	2.2E-94	1.3E-58	4.2E-09	2.9E-13	3.8E-163	3.8E-163	~ 0
PEARSON	2.3E-144	3.2E-139	2.4E-105	1.7E-86	2.5E-26	7.5E-46	1.4E-166	1.4E-166	~ 0
PHI GEOMETRICAL MEAN	2.4E-144	3.2E-139	2.4E-105	1.7E-86	3.0E-50	2.7E-51	1.4E-166	1.4E-166	~ 0
ROGERS AND TANIMOTO	3.0E-52	4.4E-24	1.3E-140	2.8E-147	0.1687	0.2596	1.4E-166	1.4E-166	~ 0
ROGOT1	1.4E-123	2.5E-100	4.7E-29	0.99999	5.8E-26	3.3E-08	1.4E-166	1.4E-166	~ 0
ROGOT2	2.4E-144	3.5E-139	2.4E-105	1.7E-86	4.9E-49	4.2E-51	1.4E-166	1.4E-166	~ 0
RUSELL AND RAO	5.0E-138	1.8E-128	7.3E-142	9.2E-148	8.1E-09	3.3E-14	1.4E-166	1.4E-166	~ 0
SCOTT	1.4E-123	2.5E-100	4.7E-29	0.99999	1.7E-26	6.2E-09	1.4E-166	1.4E-166	~ 0
SIMPLE MATCHING	3.0E-52	4.4E-24	1.3E-140	2.8E-147	0.1687	0.2596	1.4E-166	1.4E-166	~ 0
SOKAL	3.0E-52	4.4E-24	1.3E-140	2.8E-147	0.1687	0.2596	1.4E-166	1.4E-166	~ 0
SORENSEN DICE	6.6E-140	3.9E-131	1.8E-142	2.8E-147	4.4E-53	3.3E-55	1.4E-166	1.4E-166	~ 0
TARANTULA	1.2E-129	1.6E-124	8.9E-125	6.9E-100	2.8E-36	6.0E-39	2.8E-166	2.8E-166	~ 0
TARWID	1.2E-129	1.6E-124	1.3E-115	1.3E-68	7.8E-36	7.2E-39	2.8E-166	2.8E-166	~ 0
WONG1	5.0E-138	1.8E-128	7.3E-142	9.2E-148	8.1E-09	3.3E-14	1.4E-166	1.4E-166	~ 0
WONG2	3.0E-52	4.4E-24	1.3E-140	2.8E-147	0.1687	0.2596	1.4E-166	1.4E-166	~ 0
WONG3	3.7E-147	7.7E-142	2.5E-142	2.8E-147	6.6E-38	9.0E-51	1.4E-166	1.4E-166	~ 0
WONG3 PRIME	3.7E-147	7.7E-142	1.7E-142	2.8E-147	1.1E-38	2.7E-54	1.4E-166	1.4E-166	~ 0
ZOLTAR	2.4E-147	1.4E-142	1.0E-144	7.3E-148	4.8E-32	8.3E-51	1.4E-166	1.4E-166	~ 0

Table 20 AUC normalized in each program according to the ranking metrics used in fault localization

	Programs								
	Open Ankus		Data quality analysis		Movies analysis		Data cleaner Knn analysis		All programs
Ranking metric	[1-10]	[11-35]	[1-10]	[11-35]	[1-10]	[11-35]	[1-10]	[11-35]	
M2	0.985	0.958	0.997	1.000	0.832	0.841	1.000	1.000	0.917
O POW P	0.984	0.962	0.998	1.000	0.819	0.837	1.000	1.000	0.915
WONG3 PRIME	0.982	0.961	0.994	0.999	0.819	0.838	1.000	1.000	0.914
KULCZYNSKI2	0.977	0.956	0.996	0.999	0.823	0.835	1.000	1.000	0.914
MCCON	0.977	0.956	0.996	0.999	0.823	0.835	1.000	1.000	0.914
WONG3	0.982	0.961	0.994	0.999	0.817	0.833	1.000	1.000	0.914
ZOLTAR	0.985	0.963	0.996	0.999	0.807	0.833	1.000	1.000	0.913
O	0.985	0.961	0.997	1.000	0.799	0.828	1.000	1.000	0.912
OCHIAI1	0.960	0.938	0.996	0.999	0.830	0.832	1.000	1.000	0.910
FOSSUM	0.967	0.909	0.997	1.000	0.826	0.831	1.000	1.000	0.907
MINUS	0.978	0.959	0.957	0.941	0.830	0.840	1.000	1.000	0.904
ANDERBERG	0.931	0.899	0.994	0.999	0.838	0.831	1.000	1.000	0.903
DICE	0.931	0.899	0.994	0.999	0.838	0.831	1.000	1.000	0.903
GOODMAN	0.931	0.899	0.994	0.999	0.838	0.831	1.000	1.000	0.903
JACCARD	0.931	0.899	0.994	0.999	0.838	0.831	1.000	1.000	0.903
KULCZYNSKI	0.931	0.899	0.994	0.999	0.838	0.831	1.000	1.000	0.903
SORENSEN DICE	0.931	0.899	0.994	0.999	0.838	0.831	1.000	1.000	0.903
MOUNTFORD	0.941	0.918	0.996	0.999	0.810	0.814	1.000	1.000	0.901
AMPLE2	0.963	0.943	0.949	0.933	0.850	0.833	1.000	1.000	0.900
BRAUN BANQUET	0.899	0.869	0.994	0.999	0.848	0.835	1.000	1.000	0.897
PHI GEOMETRICAL MEAN	0.956	0.939	0.957	0.941	0.832	0.824	1.000	1.000	0.897
HARMONIC MEAN	0.956	0.939	0.957	0.941	0.830	0.823	1.000	1.000	0.897
ROGOT2	0.956	0.939	0.957	0.941	0.830	0.824	1.000	1.000	0.897
PEARSON	0.956	0.939	0.957	0.941	0.793	0.820	1.000	1.000	0.892
OCHIAI2	0.923	0.898	0.957	0.941	0.853	0.827	1.000	1.000	0.891
BARONI URBANI AND BUSER	0.887	0.855	0.994	0.999	0.833	0.819	1.000	1.000	0.891
ARITHMETIC MEAN	0.920	0.899	0.957	0.941	0.812	0.818	1.000	1.000	0.885
DENNIS	0.918	0.898	0.954	0.931	0.818	0.811	1.000	1.000	0.883
RUSELL AND RAO	0.908	0.879	0.992	0.998	0.740	0.757	1.000	1.000	0.876
WONG1	0.908	0.879	0.992	0.998	0.740	0.757	1.000	1.000	0.876
BINARY	0.908	0.880	0.989	0.997	0.739	0.752	1.000	1.000	0.875
CBI SQRT	0.891	0.878	0.950	0.930	0.790	0.793	1.000	1.000	0.871
CBI LOG	0.888	0.838	0.950	0.931	0.801	0.799	1.000	1.000	0.868
COHEN	0.859	0.847	0.956	0.941	0.800	0.799	1.000	1.000	0.868
MICHAEL	0.985	0.961	0.928	0.893	0.862	0.841	0.857	0.857	0.866
FLEISS	0.864	0.778	0.976	0.982	0.782	0.730	1.000	1.000	0.857
TARANTULA	0.876	0.863	0.972	0.953	0.801	0.801	0.918	0.918	0.856
TARWID	0.876	0.863	0.969	0.943	0.800	0.800	0.918	0.918	0.855
CBI INCREASE	0.876	0.859	0.969	0.943	0.800	0.803	0.918	0.918	0.854
SCOTT	0.872	0.806	0.917	0.852	0.785	0.737	1.000	1.000	0.839
ROGOT1	0.872	0.806	0.917	0.852	0.784	0.735	1.000	1.000	0.838
OVERLAP	0.827	0.804	0.954	0.937	0.738	0.751	0.919	0.919	0.826
AMPLE	0.882	0.851	0.880	0.886	0.786	0.717	0.857	0.857	0.809
EUCLID	0.672	0.594	0.992	0.999	0.699	0.693	1.000	1.000	0.803
HAMANN	0.672	0.594	0.992	0.999	0.699	0.693	1.000	1.000	0.803
HAMMING	0.672	0.594	0.992	0.999	0.699	0.693	1.000	1.000	0.803
M1	0.672	0.594	0.992	0.999	0.699	0.693	1.000	1.000	0.803
ROGERS AND TANIMOTO	0.672	0.594	0.992	0.999	0.699	0.693	1.000	1.000	0.803
SIMPLE MATCHING	0.672	0.594	0.992	0.999	0.699	0.693	1.000	1.000	0.803
SOKAL	0.672	0.594	0.992	0.999	0.699	0.693	1.000	1.000	0.803
WONG2	0.672	0.594	0.992	0.999	0.699	0.693	1.000	1.000	0.803
GOWER	0.615	0.562	0.909	0.923	0.534	0.558	1.000	1.000	0.734
RANDOM	0.474	0.469	0.872	0.872	0.678	0.677	0.505	0.505	0.614

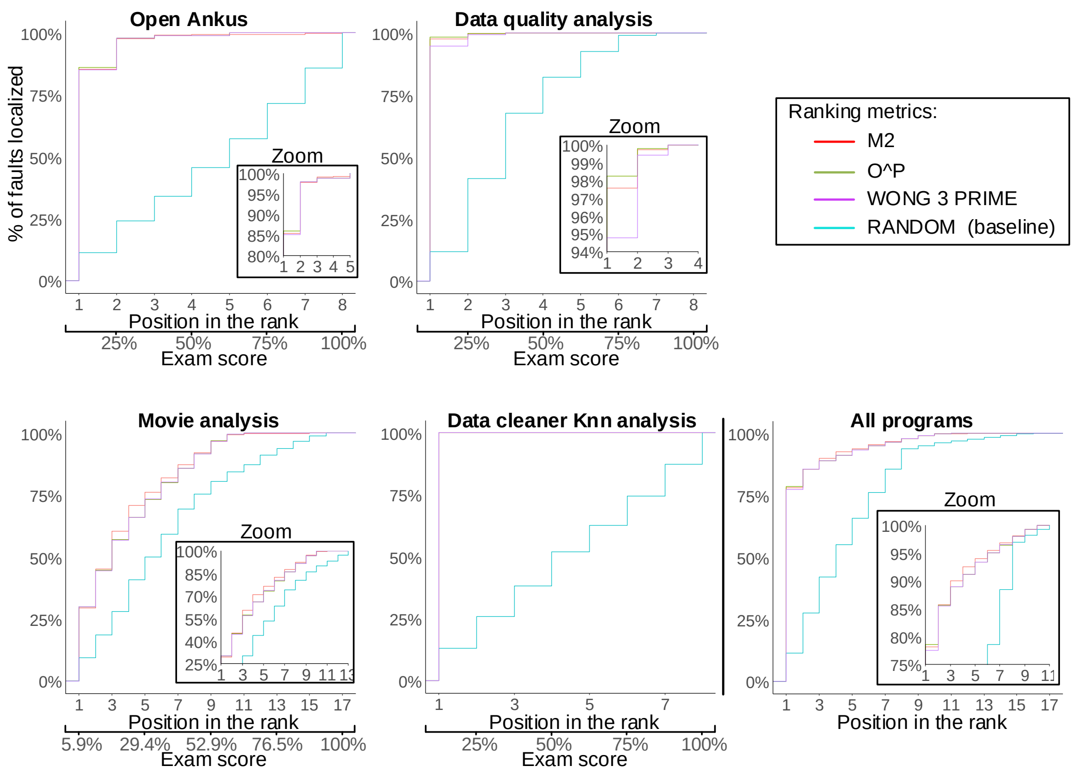

Fig. 28 Percentage of accumulated fault rightly localized in each position of the ranking for all programs according to different ranking methods

The experiments demonstrate that the fault localization technique is effective locating the root cause of the faults. Other goal of the experiments is also to evaluate the efficiency. Fig. 29 describes the execution time according to the number of <key, value> pairs of the test case. The localization technique employs few time, generally less than 6 seconds. The execution time also follows a linear trend increasing the seconds when the input data is bigger. Then **the answer of RQ10 is that the fault localization technique is efficient employing only few seconds and is also scalable with respect to the size of the test input data**.

The fault localization technique obtains the root cause of the fault by means of the analysis of several configurations. Depends on the number of configurations, the technique could yield

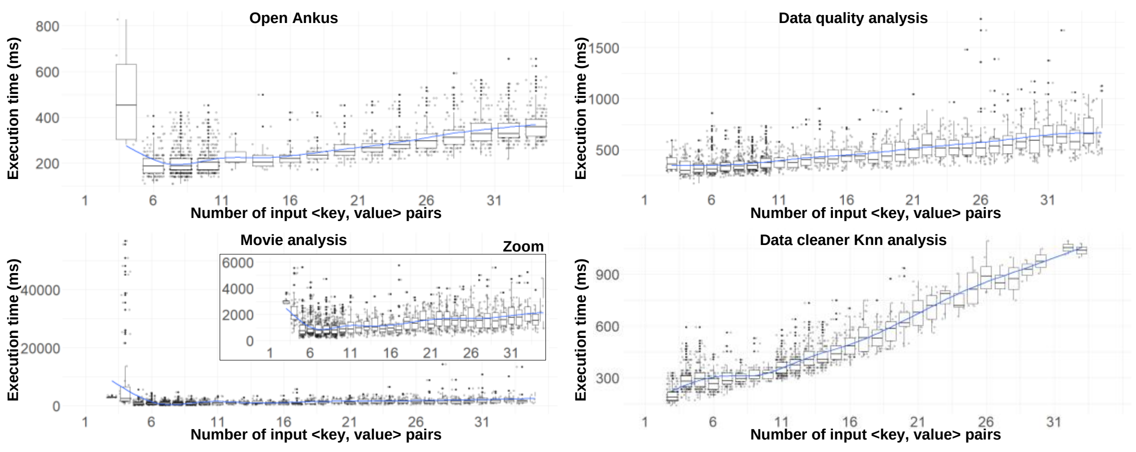

Fig. 29 Execution time of the fault localization technique according to the number of <key, value> pairs

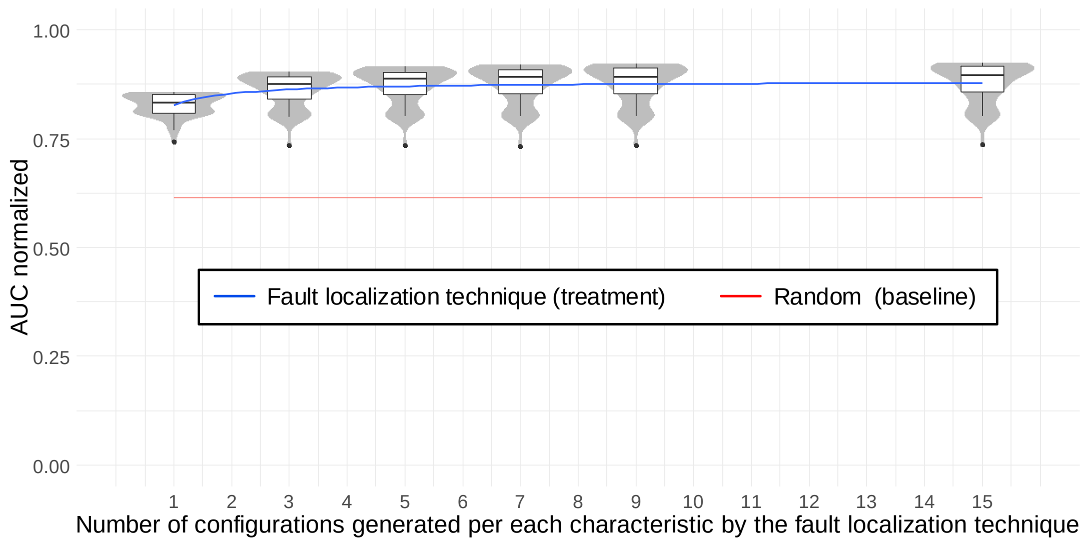

Fig. 30 Effectiveness of ranking metrics used by the fault localization technique according to the number of configurations generated

better or worst results. To obtain the ideal number of configurations, Fig. 30 summarizes the trend of the distribution of the AUC normalized in different ranking metrics along all experiments varying the number of configurations. As expected, the fault localization obtains better results when the number of configurations generated/analyzed increases, but also decrease the efficiency. To obtain the number of configurations beyond which there is no improvement in AUC, we calculate the second derivate of the curve. In this derivate, there is no value equally to zero, then the AUC normalized is always slightly increased until a limit of 0.88. This improvement is bigger between 1 to 5 number of configurations (from 0.82 to 0.87) and very lower after that (from 0.87 to 0.88). Then **the answer of RQ11 is 5 number of configurations** because, as we can see in RQ7-RQ10, yields to a very good effectiveness and very good efficiency.

IV.4.2 Input Reduction experiments

The *goal* of this experiment is the assessment of how effective and efficient the input reduction techniques are isolating the data that triggers the *MapReduce* design faults. The research questions are:

RQ12. Are both input reduction techniques, genetic algorithm and delta debugging, better search strategies than random searches (baseline) to isolate the input data that trigger the *MapReduce* design faults?

RQ13. Which of the input reduction techniques proposed, genetic algorithm and delta debugging, is better isolating the data that trigger the *MapReduce* design faults?

RQ14. How much data reduce the input reduction techniques?

RQ15. How much execution time is employed by the input reduction techniques of the *MapReduce* design faults?

The setup of the experiments aimed to answer the previous research questions is described in Subsection IV.4.2.1. The results are detailed in Subsection IV.4.2.2.

IV.4.2.1 Input Reduction experiments: setup

The experiments are very similar to the fault localization experiments. The same test cases that triggers the faults are used as *experimental units*, but only in those with 11-35 <key, value> (1000 test case per each of the 4 programs). The reason to not include in the experiment the test cases with 1-10 <key, value> pairs is because the search space is small (lower than 1024 subsets of input data) and then can be analyzed thoroughly without a search strategy. The search space grows exponentially, and then is not viable to explore thoroughly all search space with 11-35 <key, value> pairs due costly reasons (search space from 2047 to 34359738367 subsets of input data). In these test cases, the search strategies proposed in this chapter can be used.

In this experiment, the *dependent variable* or *response variable* is the quantity of data reduced/isolated (RQ12-RQ14) and the execution time (RQ15). The *baselines* of the experiment are random searches in the search space (RQ12) and delta debugging (RQ13). The random baseline aims to find a minimal subset of the input data that trigger the fault by means of random searches in the space, that is, the technique selects randomly without substitution several subsets of the input data. In the experiment, two random baselines are used, Random-G and Random-D, that perform per each test case the same number of searches than genetic algorithm and delta debugging, respectfully. The *treatments* of the experiment are the genetic algorithm (RQ12-RQ13) and delta debugging (RQ12). The genetic algorithm is parametrized with the default options: initial population of 20 individuals, the asexual reproduction generates 4 individuals in each generation and the mutation probability is 1%.

As in the fault localization experiment, the research question related to the efficiency, RQ15, is answered analysing the execution time trend. The other research questions are about effectiveness and are answered according to the data reduced. The RQ14 is answered analysing the percentages reduced by the input reduction techniques. The other two research questions, RQ12-RQ13, are answered with statistical tests as in other test suite reduction papers [232]–[234]. The difference is that the techniques of this chapter instead to reduce the size of the test suite, are focused in the reduction of the input data of the test cases. Then instead to measure the percentage of test suite reduced [235], [236], the experiment measures the percentage of test input data reduced. The comparison of the percentages of input data reduced is done by the t test *statistical test* that measures the differences among the paired means with the following one-tail null hypothesis: H_{02}: mean(% reduced by genetic algorithm) = mean(% reduced by Random-G) (RQ12), H_{03}: mean(% reduced by delta debugging) = mean(% reduced by Random-D) (RQ12), and H_{04}: mean(% reduced by genetic algorithm) = mean(% reduced by delta debugging) (RQ13).

IV.4.2.2 Input Reduction experiments: results

Fig. 31 details the distribution of the percentage of data reduced by each input reduction technique. Table 21 also summarizes the mean of percentage reduced. The genetic algorithm and delta debugging are clearly better techniques than their random counterpart even though, respectively, they perform the same number of searches. The p-values are lesser than 0.001, then the null hypothesis H_{02} and H_{03} are rejected. **The answer of RQ12 is that both, the genetic algorithm and delta debugging, reduce better the test input data than their equivalent random techniques**.

Both techniques, genetic algorithm and delta debugging achieve in the *Data cleaner Knn analysis* program the same percentage of input reduction (85.4%). The data of this program is easy to

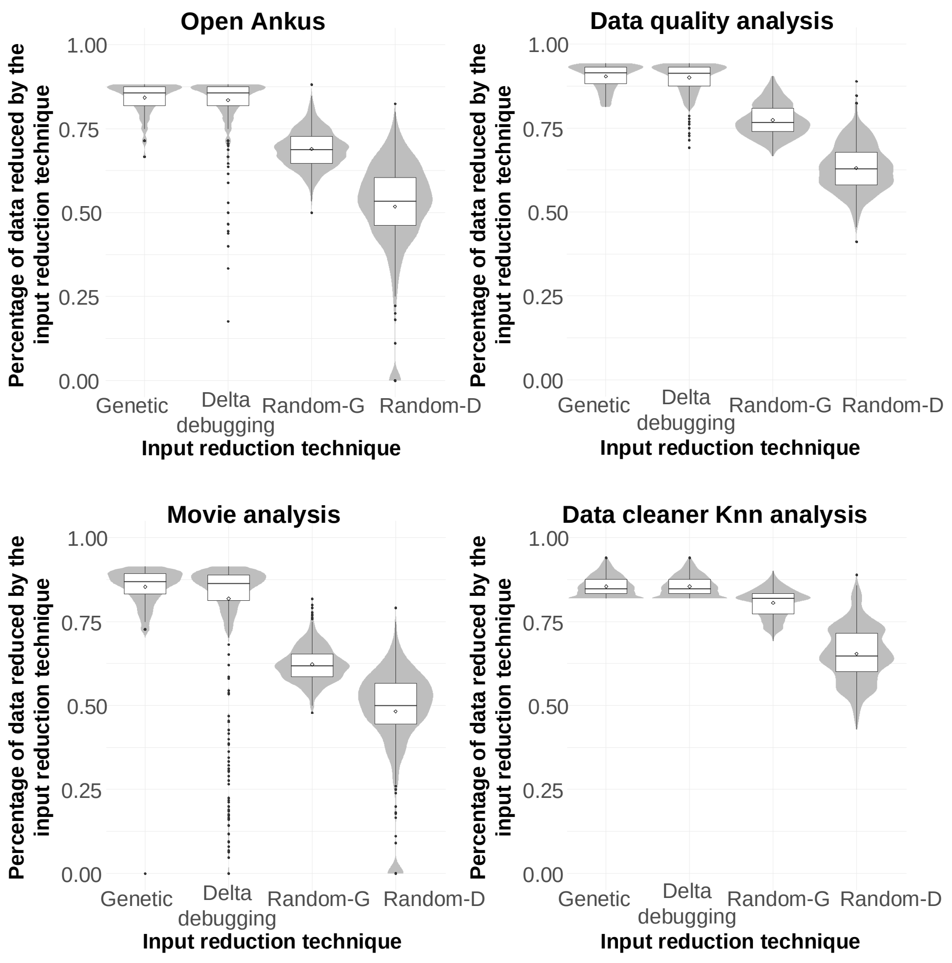

Fig. 31 Distribution of percentage of input reduction in each program by different input reduction techniques

reduce because also by a random search can be achieved good reduction (80.5% by Random-G). Then the search space of this program contains a lot of good/optimal solutions and both techniques, genetic algorithm and delta debugging, reach them easily. For the other 3 programs, the search space is more complex and the selection of a properly search strategy could yield in better input reduction. In these programs, the genetic algorithm achieves significant better input reduction than delta debugging. For the *Open Ankus* program, genetic algorithm reduces in average 84.3% of input data and delta debugging 83.5% (p-value < 0.001); for the *Data quality analysis*, 90.4% and 90% (p-value < 0.001); and for *Movie analysis*, 85.4% and 81.9% (p-value < 0.001). The null hypothesis H_{04} is rejected. Then **the answer of RQ13 is that the genetic algorithm is significant better isolating the input data that trigger the fault than delta debugging**.

Despite the genetic algorithm is significant better reducing the data than delta debugging, both techniques achieve a good percentage of reductions. Table 21 and Fig. 31 indicate that both techniques usually reduce the data more than 80%-85%. Then **the answer of RQ14 is that both techniques, genetic algorithm and delta debugging reduces the majority of the data**, but the genetic algorithm achieves significant better rates of reduction.

Whereas delta debugging is a greedy approach that search for the first minimal data that triggers the fault, the genetic algorithm additionally tries to skip the local minimal aimed to achieve the best solution (global minimal data that trigger the fault). Then delta debugging employs few time

Table 21 Mean of the percentage of input data reduced by each technique in the programs

Input reduction technique	Program: Open Ankus	Data quality analysis	Movie analysis	Data cleaner Knn analysis
Genetic	84.3%	90.4%	85.4%	85.4%
	***	***	***	ns
Delta Debugging	83.5% ***	90.0% ***	81.9% ***	85.4% ***
	***	***	***	***
Random-G	69.0%	77.4%	62.3%	80.5%
Random-D	51.8%	63.1%	48.3%	65.3%

ns p-value > 0.05 * p-value ≤ 0.05 ** p-value ≤ 0.01 *** p-value ≤ 0.001

to isolate the data, while the genetic algorithm takes more time trying different solutions until converge. Fig. 32 details the execution time trend of each technique according to the size of input data, and Table 22 summarizes the average execution time. As expected, delta debugging employs usually few seconds to isolate the input data that trigger the fault. On the other hand, the genetic algorithm employs a lot of more time, sometimes in hour scale. The other disadvantage of the genetic algorithm is that the execution time grows faster according to the number of <key, value> pairs to be isolated. In contrast, the execution time of delta debugging grows more slowly. Then **the answer of RQ15 is that whereas delta debugging is scalable employing few seconds, the genetic algorithm does not scale rightly and employs minutes-hours**.

IV.4.3 Discussion

The experiments indicate that *MapReduce* programs with design faults can be automatically debugged with the fault localization and input reduction techniques proposed. The fault localization discovers automatically in few seconds the right root cause of the design faults, specially when employs the M2, O^P or Wong3 prime ranking metrics. In the other hand, the input reduction techniques, genetic algorithm and delta debugging, isolate automatically the data that trigger the fault achieving usually more than 80-85% reduction of the input data. The genetic algorithm obtains significant better reductions than delta debugging, but also employs

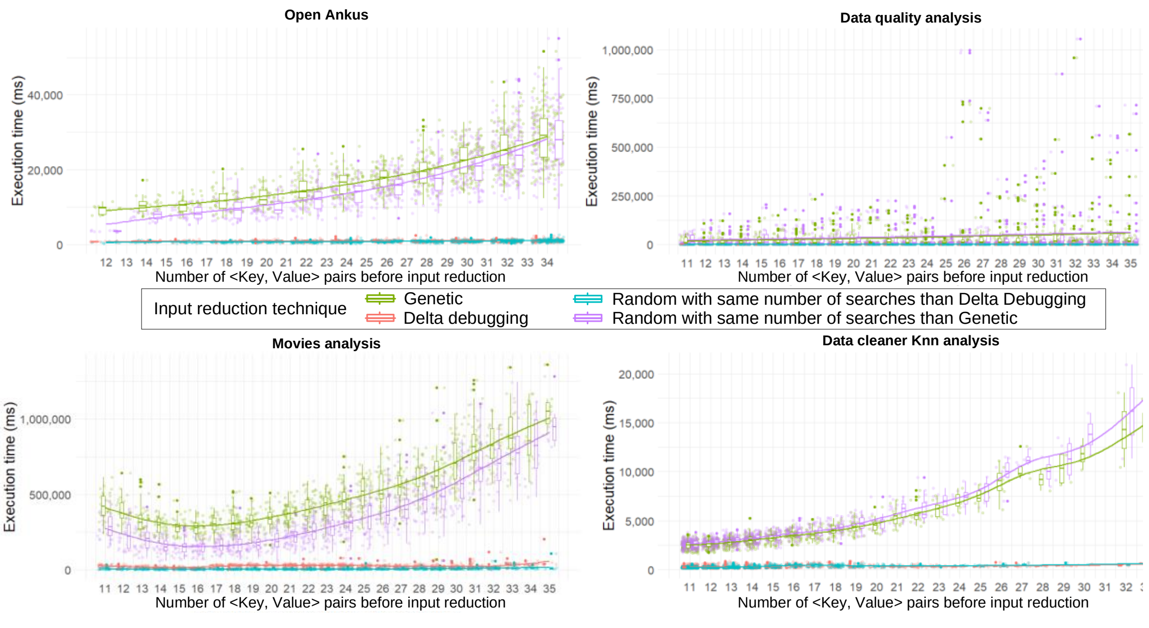

Fig. 32 Execution time of the test input reduction according to the number of <key, value> pairs

Table 22 Mean of the execution time of input reduction measured in milliseconds by each technique in the programs

Input reduction technique	Program			
	Open Ankus	**Data quality analysis**	**Movie analysis**	**Data cleaner Knn analysis**
Genetic	20195 ms	3632696 ms	484364 ms	3341 ms
Delta Debugging	898 ms	57079 ms	25634 ms	306 ms
Random-G	18337 ms	4132114 ms	341645 ms	3570 ms
Random-D	811 ms	44829 ms	6361 ms	303 ms

more time. Then the genetic algorithm can be used to reduce the input data during laboratory/offline testing because the execution time is not an issue. In contrast, the execution time is an issue in the runtime testing like the approach proposed in the next chapter, then in this scenario is better to use delta debugging because in few seconds achieves good reductions. The remainder of this subsection discusses the limitations of these experiments through the threats of validity and their subcategories [194], [198], [199].

The **conclusion** threats are those issues that could affect the concluding of the experiments. The debugging techniques of this chapter generate test cases with different configurations and then take advantage of the testing technique proposed in the previous chapter to automatically check if there is a failure or not. This oracle is not 100% accurate and could provide wrong information for the debugging techniques (*inaccurate data*). Then the fault localization technique and the input reduction techniques (genetic algorithm, delta debugging, and also the baselines Random-{G,D}), decrease their effectiveness. This is not a big issue because all techniques uses the same oracle, and usually provides the right information [41].

The **internal** threats are those issues regarding the causal relationship between independent variables and dependent variables. RQ10 and RQ15 analyze the execution time, but some noise can be introduced into the measurements by other operative system tasks (*confounding effects of variables*). To mitigate this problem, the experiments are executed in the same computer without any other programs operating in the background.

The tool that automates the research, MRDebug, can contain faults and other limitations. To mitigate the potential faults of the tool, both manual and automatic testing was performed mainly from the functional and performance point of view. This tool also uses MRTest as testing technique/oracle and has the same limitations described in the previous chapter (Section III.4.3). For example, when the program under test inserts data in an external database, MRDebug and MRTest can perform the insertions for each of the configurations generated/executed. When the external service is fully controllable, then the tester can handle these side effects inside the test cases.

The **external** threats are those issues that can affect the generalization of the results. The subjects of this experiment are 8000 test cases randomly selected from 4 *MapReduce* programs selected by consecutive sampling. Ideally, the programs should also be selected randomly, but often this is not feasible in software engineering (*Interaction of selection and treatment*). For *Big Data* programs, there is no benchmark of faults and industrial programs are not usually available. This problem is mitigated by using some real-world applications, instead of using programs with seeded faults (hand-seeded faults or mutation faults) that are prone to other external threats [200], [201]. Therefore, there are other issues regarding seeded faults when they are used to evaluate testing techniques. The hand-seeded faults are injected by the expert and they are subjective, decrease the reproducibility of the experiments and are not representative of real faults in terms of easy detection [202]. In contrast, mutation faults are representative of the majority of faults, but this is not the case when the developer implements an incorrect algorithm [203]. The faults pursued by this thesis fall into the previous category of faults that are not possible to substitute with mutations. The faults that are the target of this thesis are caused by incorrect design decisions that lead to the implementation of faulty algorithms, completely different from those of the correct implementation. As such, the injection of mutation faults is not a feasible way to evaluate the debugging techniques of this chapter.

The test cases used to evaluate the debugging techniques has 1-10 and 11-35 input data. The results of the experiments could not be generalizable for all other sizes of the input data (*Applicability of results across different samples*). However, in the experiments the debugging techniques scale rightly regardless of the input data. The sizes of the input data used in the experiments (1-10 and 11-35) follows the same design of our previous experiments [41]. During functional testing and debugging, when the test cases has more data, more difficult to both detect and debug the faults. Then, the functional test cases are usually designed with few data, specially in when the programs are executed in distributed fashion.

Other results can be obtained if the fault localization technique generates/analyzes the configurations in a different way, or if the input reduction technique employs other search strategy (*Applicability of results when technique is varied*). Despite there are room to improve, the techniques achieve very good effectiveness.

The **construct** threats are those issues between the experiment and its underlying theoretical concepts. The fault localization technique is only compared against a random localization because the other techniques of the literature are not suitable for *MapReduce* design faults. These other techniques are usually focused in the analysis of the statements instead of configurations. In the case of the input reduction, the genetic algorithm is compared against both random and an adaptation of the delta debugging algorithm that is very used to isolate faults.

One part of the experiment analyses the efficiency of the debugging techniques based only on the execution time measure, but there could be more measures not considered, such as memory (*Mono-operation bias*). To mitigate this problem, the experiments were executed in a commodity computer with few resources. The memory does not appear relevant because its usage was low during the experiments. Furthermore, the tool that automates the research was tested to avoid memory bottlenecks.

IV.5 Related Work

Debugging distributed programs is a difficult task, specially in the *Big Data* field [237]. Several works propose debugging techniques focused on performance for *Big Data* frameworks [84], [238] and others for the *MapReduce* programs [239], [240]. In contrast, the current chapter is not focused on performance debugging, but on functional debugging. Olston et al. [241] interview ten employers of Yahoo! about debugging dataflow programs like *MapReduce*. The majority of them suggests that can be valuable to obtain the data and operators that cause the failure. The current chapter undertakes both tasks in *MapReduce* design faults through the

MRDebug framework. MRDebug localizes the root cause of the fault, isolates/reduces the data that trigger the failure, and supports the common debugging utilities such as breakpoints.

Fault localization: Daphne [242] is a debugger for DryadLINQ (framework that supports and extends the *MapReduce* processing model). This debugger diagnostics the root cause of the faults based on a decision tree at different levels of abstraction considering logs and stack traces of the execution. The current work does not analyze neither logs nor stack traces because is focused on the failures that are triggered by some non-deterministic executions. Then MRDebug analyzes with spectrum-based fault localization not only one execution, but several executions to localize the non-deterministic characteristics that trigger the failure.

Isolation/reduction of the data: BigSift [243] is a runtime debugger for applications executed in Spark (framework that support and extends the *MapReduce* processing model). This debugger isolates the data through delta debugging combined with data provenance and an automatic oracle provided by the tester. The approach proposed in the current chapter also isolates the data through delta debugging but do not need that the tester provides an oracle. MRDebug is focused only on design faults and then uses a generic oracle from our previous chapter (MRTest). In addition to delta debugging, the current work also performs the isolation of the data through a more generalized search-based algorithm [244].

Debugging utilities: Inspector Gadget [241] is a debugger that alerts about predicate violations and also traces the data that produce the failures in Pig programs (abstraction of the *MapReduce* processing model). Our previous work also alerts of potential failures in production [52], but only for those caused by *MapReduce* design faults. The current work, MRDebug, also allows to trace the failures, but only for those caused by design faults and only at high level with breakpoints and watchpoints in a simulation environment.

BigDebug [245] is a runtime debugger that allows to insert simulated breakpoints and watchpoints in Spark production environment. In contrast, the approach proposed in the current chapter simulates the production environments to allow the insertion of the breakpoints and watchpoints.

Other works are focused on record and replay failures. Arthur [246] is a debugger for Hadoop and Spark that traces the relevant data and allows to replay the failure. Newt [247] is another debugger of *MapReduce* applications that captures runtime information allowing the tracing and reproduction of failures. Bergen et al. [248] proposes a debugger for Spark that records failures from production and reproduces these failures locally to support breakpoints. These approaches that are focused on record-replay failures does not handle properly the non-determinism failures, that is the main goal of the current work. Arthur [246] considers a checksum of the output and then can detect non-determinism, but is not able to reproduce non-deterministic results. Newt [247] can also record the non-deterministic data but is not able to reproduce them deterministically. The current work, MRDebug, is focused in debugging failures caused by non-deterministic executions due a design fault. MRDebug simulates different infrastructure configurations to capture several executions that cause the failure and several that mask it. Then MRDebug can reproduce deterministically the non-deterministic executions through seeds.

IV.6 CONCLUSIONS

The design faults of the *MapReduce* can be automatically debugged to locate the root cause of the fault and isolate the data that trigger the failures. The common fault localization techniques are focused on the source code, but these *MapReduce* faults are caused by wrong design instead of the code. The root cause of these faults can be automatically obtained through spectrum-based fault localization analysing the infrastructure configurations instead of the source code.

The root cause of the fault is not usually enough to understand the fault. In order to improve the debugging, the isolation/minimization of the data that trigger the failure can help to understand the fault. The relevant data can be isolated through search-based algorithms aimed to reduce the data that triggers the failure. The localization and input reduction techniques are automatized in a debugging framework called MRDebug. This framework also simulates the distributed executions to support the common debugging utilities such as breakpoints and watchpoints.

We performed an empirical study to analyze the effectiveness and efficiency of both localization and input reduction techniques. The results showed that both techniques debug rightly the programs in reasonable time. The localization technique obtains the root cause of the faults analysing few configurations in scalable way employing also few seconds. The spectrum-based fault localization techniques use a similarity metric to analyze the potential root causes of the fault. MRDebug obtains usually the right root cause of the fault regardless of the similarity metric employed, but the bests for *MapReduce* design faults are M2, O^P and Wong3 prime.

MRDebug contains two input reduction techniques: delta debugging and genetic algorithm. Both techniques reduce the majority of the data making the wrong execution of the *MapReduce* design fault simpler to understand. The genetic algorithm isolates the data significantly better than delta debugging, but also employs much more time. Then the genetic algorithm is better when there are no deadlines like in offline testing on laboratory. Contrary, delta debugging is better when there are deadlines like in online testing on production as in the approach proposed in the next chapter.

V OPERATIONS

The previous chapters of the thesis propose techniques to test and debug the functionality of the *MapReduce* applications. These techniques analyze the execution of the input data under different infrastructure configurations, then both techniques are able to detect and diagnose the design faults automatically given only the input data. These input data are provided by the tester, but this chapter proposes an autonomous approach to take the data from runtime to test-debug, without human intervention, the programs that are executed in production.

Once the data are taken automatically from runtime as test input data, the testing and debugging techniques can detect and diagnose the faults also in automatic way. In *Big Data* field, it is not feasible to test-debug the programs with all runtime data because is expensive from the performance and resources point of view. As initial work, we proposed a data-flow criteria called MRFlow [53] that is adapted to *MapReduce* considering the different transformations of data from *Mapper* to *Combiner-Reducer*. This data-flow criteria has also several limitations to use automatically during runtime because requires to both instrument the program and analyze the whole runtime data to obtain a sample that covers the data-flow criteria. Then the data-flow criteria is not feasible at all to obtain in few seconds the input data for testing-debugging each program executed production. Instead to obtain an input data for each program like in the data-flow criteria, the data can be obtained for all programs that use the same runtime data and then use a cache mechanism to accelerate the execution of the testing-debugging techniques in few seconds.

The approach proposed in this chapter, MrExsist, identifies the programs executed in production and then takes samples of the runtime data (usually from cache) to perform testing and debugging with the techniques of the previous chapters. This approach takes both program and data from production, but instead to test-debug the applications in production (In Vivo), test-debug these applications in laboratory to obtain a fine-grained control and reproducibility of the tests. The chapter proposes a hybrid approach between testing-debugging 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 majority of this chapter is published in QRS 2017 [52]. Section V.1 discusses the background and related work about testing in production. Section V.2 defines the Ex Vivo testing approach, and Section V.3 describes the MrExist framework based on this approach that is adapted to the *MapReduce* processing model. Then MrExist is validated in Section V.4. Finally, the conclusions are included in Section V.5.

V.1 BACKGROUND AND RELATED WORK

In the production environment, the infrastructure failures are fairly frequent. Several research lines suggest the injection of infrastructure failures [85], [148] during the testing, and several tools support it [64]–[66]. For example, Marynowski et al. [147] 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 like a bacteriological algorithm [149] or input domain together with combinatorial testing [172]. Unlike the previous testing techniques, this chapter takes the data directly from production in runtime.

There are several tools to design and execute test cases for *MapReduce* applications. Herriot [63] 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 [62] execute the test cases in a cluster simulated in memory. For unit testing, MRUnit [61] provides an adaptation of JUnit [60] to the *MapReduce* processing model. This chapter 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.

V.2 Ex Vivo Testing

Modern software applications are increasingly distributed, pervasive, and adaptive. For such systems, the boundary between development-time and production is vanishing [249], and several authors (e.g., [250], [251]) 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 [252]:

- 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 [253]. 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 [254]. The information of these datasets can be useful for carrying out testing in runtime [255], 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.

Thus, this chapter 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. 33 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.3 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 V.2 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.

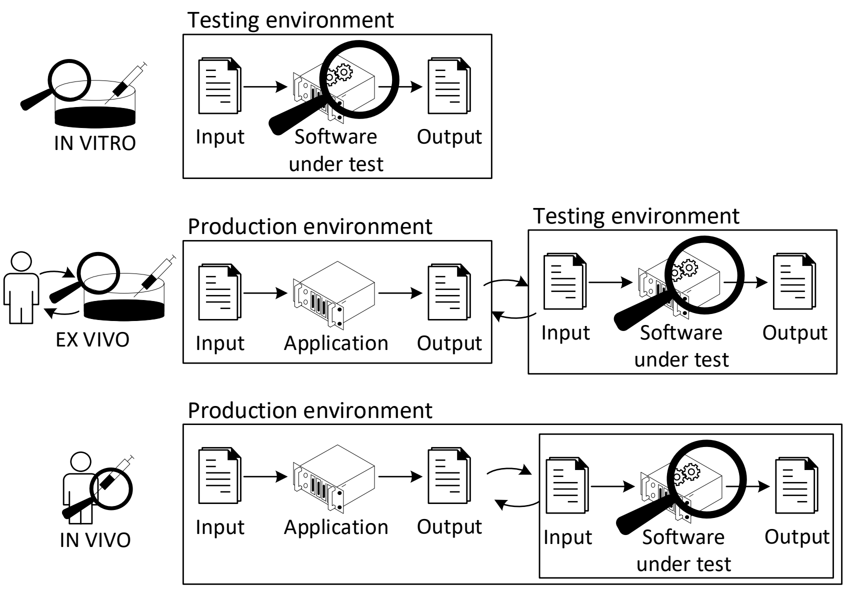

Fig. 33 Software testing approaches

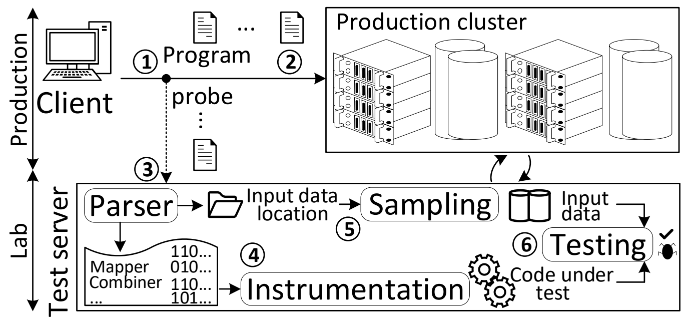

Fig. 34 Architecture of MrExist (MapReduce EX vivo teSTing)

Therefore the execution of the test cases does not take resources from production, does not introduce side-effects and can be fully controlled.

Fig. 34 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 the testing technique of the chapter III that only needs the test input data and the program to detect functional faults (6). Once the faults are detected, the debugging technique of the previous chapter can be integrated automatically.

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 [181], as for example some machine learning programs. Such problems do not exist in those faults addressed by the thesis, as we can compare the outputs obtained for the same data in different configurations (Chapter III). 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 Fig. 10 (Chapter III). 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. This program optimizes the performance through a *Combiner* functionality that 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 [42] 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.

V.3.1 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 [256] 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 [257] 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.

V.3.2 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 [258] such as Mockito [259] and PowerMock [260].

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.

V.3.3 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 [261] or HDFS (Hadoop Distributed File System) [262]. 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 [263]. 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 [264]. 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 [28], and sometimes concurrently [265]. To avoid multiple samplings of these *Big Data* datasets a cache is implemented to improve performance [266], [267]. 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 the following 20 temperatures also depicted in the top of Fig. 20: year 1995 with 7º, 9º, 7º and 10º; year 1996 with 1º, 3º, 2º and 5º; year 1997 with 8º and 6º; year 1998 with 5º; year 1999 with 4º, 2º and 3º; year 2000 with 5º and 10º; year 2001 with 9º and 8º; year 2002 with 12º, and year 2003 with 13º. Then these test input data are available in the cache for future uses in testing.

V.3.4 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 as happen in those programs of the previous chapters.

MrExist detects these faults employing a specific *MapReduce* testing technique proposed in Chapter III (MRTest). 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. 35 describes the execution of the testing technique taking advantage of the sampling and instrumentation of the previous subsections. 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 [42], [61] (JUnit [60] for *MapReduce*), and checked with Hamcrest matchers [268].

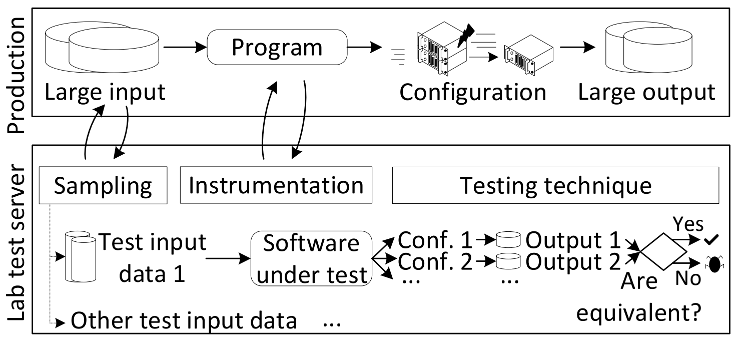

Fig. 35 Testing technique used in MrExist

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. The top of Fig. 20 describes the testing performed with 20 test input data using MRTest testing technique (Chapter III). 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*. The second configuration executes the same data with concurrency and different optimizations obtaining different outputs than the previous configuration. Then a fault is automatically discovered in the program executed in production and the developer can debug this program automatically using the MRDebug debugging approach (Chapter IV). 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.

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 middle of Fig. 20. Then the user can debug and 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 [28] or days [27].

V.3.5 Test oracle

The test oracles have some properties to characterize the testing efficacy [182], [183]. The MrExist framework aims to detect faults without human intervention, and the oracle used during testing is an automated partial oracle [181]. 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 [184] using metamorphic testing [45], [269], [270], that is a field also employed to test machine learning programs [185] and in In Vivo frameworks [271]. 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 [186]. 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 [272].

According to the study of Segura et al. [186] 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 chapter 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.

V.3.6 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 [273] and serverless database SQLite [274], [275] 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.

V.4 CASE STUDY

In order to validate the testing framework MrExist, we use the real-world program Open Ankus [170] as case study. This program is also used to evaluate the other techniques proposed in the thesis (chapters III and IV). Open Ankus implements both 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. 36 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 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.

V.5 Conclusions

This chapter 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 the thesis 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 debug and stop the

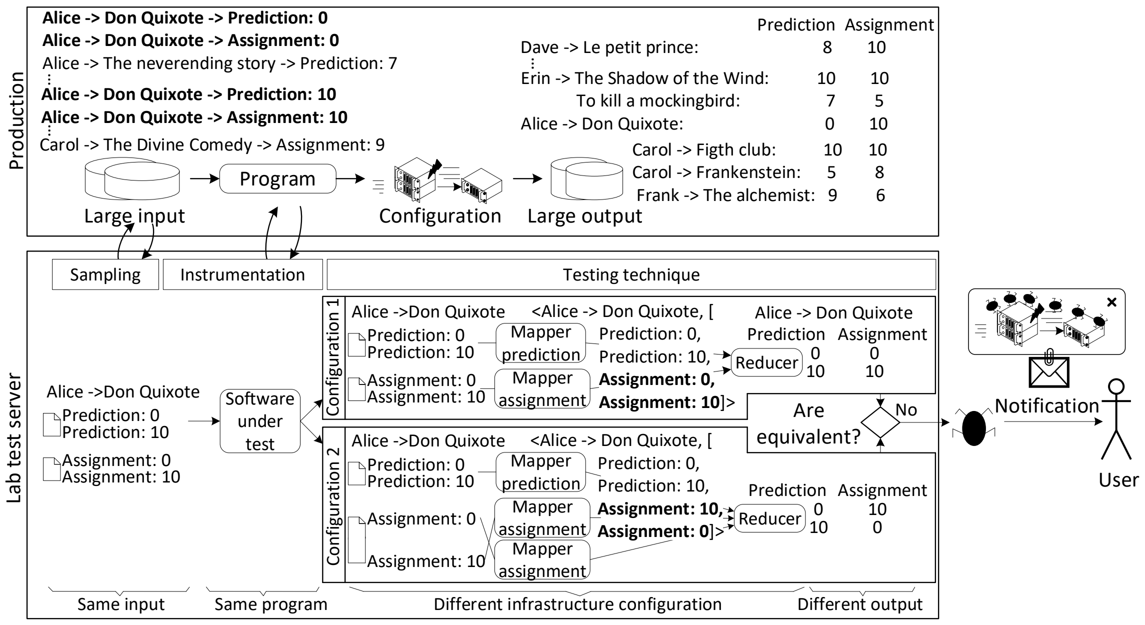

Fig. 36 Fault detected automatically by MrExist in the recommendation system of Open Ankus

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.

VI FINAL REMARKS

This thesis enhances the state-of-the-art of the software testing and debugging in the *MapReduce* field proposing new techniques that support the following hypothesis (Section I.2):

H1: The *MapReduce* applications have specific characteristics that another kind of applications do not have, such as delegate their execution to a framework that handles the massive execution splitting the datasets along several servers, allocating resources in parallel, or re-executing of part of the program in case of infrastructure failures. These characteristics in conjunction are not broadly covered by the state-of-the-art testing techniques, and the *MapReduce* applications must be tested with new approaches.

H2: The functional failures of the *MapReduce* applications that are wrongly designed entail the execution of the data concurrently in several servers in non-deterministic way. These failures are not just caused by the code, but by the design. The common debugging techniques are broadly focused on the failures caused by the code but not on those caused by the wrong design, then the *MapReduce* applications must be debugged with new approaches.

The thesis is divided in four lines of research and the research questions of each one are summarized in the Section VI.1. The conclusions of this thesis are described in Section VI.2. Finally, the future work is in Section VI.3.

VI.1 Summary of Research Questions

The **first line of research** is focused on the state-of-art and challenges in software quality of *MapReduce* programs. The following research questions were answered through a systematic mapping study (Chapter II):

RQ1. Why is testing performed in *MapReduce* programs? There are at least seven reasons for testing the *MapReduce* programs. The most frequent reasons are based on performance issues (to analyze, optimize and fulfill performance goals), the existence of several or specific failures, the type and quality of the data processed by these programs, and testing to predict the resources required and efficiently select the resources to be used. To a lesser degree, the other reasons for testing are the improper use of the processing model or technology, program misconfiguration or failures after a long period of executions.

RQ2. What testing is performed in *MapReduce* programs? The majority of the research efforts in testing the *MapReduce* programs focus on the analysis of performance, and to a lesser extent the functional aspects of *MapReduce* programs.

RQ3. How is testing performed in *MapReduce* programs? Mainly by evaluation and simulation. In both cases testing is focused specifically on the *MapReduce* functions and does not consider other parts of the program. Several tools are used to perform testing, but few are available on the Internet.

RQ4. By whom, where and when is testing performed in *MapReduce* programs? Testing is mainly performed by the tester in the Software/System Qualification Testing Process and major efforts focus on the *MapReduce* program (unit and integration testing between *Map* and *Reduce* functions).

Other findings:

1. Despite several studies that are aimed at both improving and studying the state-of-art of *Big Data* technology, there are in comparison few research lines focused on software testing of the *Big Data* programs.

2. The majority of testing research in *MapReduce* applications is focused on either Map or Reduce or the integration of both, and cannot be applied to other processing models because they are specifically designed for *MapReduce*.

3. The majority of research is about performance testing, and, to a lesser degree, functional testing. This research is about verification and validation analysis, and, to a lesser degree, about dynamic testing.

4. The prediction/analysis models employed in performance testing use different numbers of heterogeneous parameters based not only on the *MapReduce* program functionality, but also on the cluster infrastructure, file system and data.

5. The most frequent reasons for testing the *MapReduce* programs are based on performance issues (analyze, optimize and fulfill performance goals), existence of several and specific failures, the type and quality of the data processed by these programs, and testing to predict and efficiently select the resources.

6. There are several rigorous reasons for testing the functionality of *MapReduce* applications, such as the percentage of programs that fail in production or the improper use of both functional semantics and data, but there are not many research efforts focused on this line of interest.

7. Whereas performance testing is done by simulation and evaluation, functional testing employs different test activities, such as static testing and structure-based testing.

8. As expected, testing research is focused on the software qualification process to help the tester.

9. The majority of research neither creates nor uses a tool for testing *MapReduce* programs.

10. Software testing research focused on *MapReduce* applications is usually published in conferences, and furthermore it is usually published without a strong validation, using only some case studies instead of rigorous empirical experiments.

According to the previous findings, the functional testing of *MapReduce* programs has several challenges. The **second line of research** faces these challenges focused on those faults caused by a wrong design of the *MapReduce* programs. These faulty programs do not support the non-determinism of the large-scale executions due the infrastructure failures or the optimizations. Then we devised a testing technique that automatically executes the test cases under different configurations that take into account this non-determinism. This technique is detailed in Chapter III and answers the following research questions:

RQ5. Do the test execution engines detect more design failures when the MapReduce test cases are executed in different configurations? Yes, the execution of the test case in different configurations reveals more design faults. Despite a random selection of configurations (MRTest-Random) can reveal several design faults, a better selection of the configurations through Partition and Combinatorial testing (MRTest-t-Wise) reveals more design faults. MRTest-2-Wise usually detects automatically the majority of design faults in MapReduce programs.

RQ6. How expensive is the execution of the test cases in several different configurations? The execution time of test cases in several configurations takes more time than execution in one configuration: x2 or x3 when a fault is detected, and x200 or x400 when a fault is not detected. Regardless of the previous differences, the execution time of the test cases in several executions is reasonable, usually few seconds.

This testing technique (MRTest) is able to detect the design faults of the *MapReduce* programs automatically without an expected output (oracle). Once the fault is detected, the next step is to debug the program to understand this fault. The **third line of research** aims to debug the *MapReduce* programs in order to (1) localize the design characteristic that triggers the failure, (2) isolate the input data that triggers the failure to reduce the irrelevant data making the test case easier to understand, and (3) allow the inspection of the parallel execution through breakpoints. These techniques are detailed in Chapter IV and answer the following research questions:

RQ7. Is the fault localization technique better providing the root cause of the MapReduce design faults than a random localization (baseline)? Yes, the localization technique achieves in general significant better results than random localization. Some ranking metrics can achieve in some programs similar and worst localization than random localization technique, but overall the fault localization technique proposed achieves the best results in the majority of ranking metrics.

RQ8. The fault localization technique obtains a ranking of the MapReduce characteristics that could be the potential root causes of the design fault. How many characteristics should be analysed until reach the root cause of MapReduce design fault? Few, the root causes of fault are in general ranked in the first positions regardless of the ranking metric used. The fault localization technique using the best ranking metrics localizes the fault usually in the first position of the ranking.

RQ9. Which ranking metrics of those used in fault localization are better to rank the root causes of the MapReduce design faults? The majority of ranking metrics achieves good results, but the best ranking metrics to locate *MapReduce* design faults are M2, O^P and Wong3 prime. These ranking metrics usually locates the fault in the first position of the ranking.

RQ10. How much execution time is employed by the fault localization of the *MapReduce* design faults? The fault localization technique is efficient employing only few seconds to localize the root cause of the fault. The test cases with more data are more difficult to debug than those with less data, but the technique also employs few seconds because is linearly scalable.

RQ11. The localization technique analyses several configurations previously generated. How many MapReduce configurations must be generated by the fault localization technique to achieve a good balance between maximum rate of design faults located and low execution time? 5. The localization technique improves the effectiveness executing more configurations, but also employs more time. From 0 to 5 configurations the effectiveness has a substantial improvement compared with the slightly improvement after 5 configurations.

RQ12. Are both input reduction techniques, genetic algorithm and delta debugging, better search strategies than random searches (baseline) to isolate the input data that trigger the MapReduce design faults? Yes, the genetic algorithm and delta debugging reduce better the test input data than their equivalent random techniques.

RQ13. Which of the input reduction techniques proposed, genetic algorithm and delta debugging, is better isolating the data that trigger the MapReduce design faults? The genetic algorithm is significant better isolating the input data that trigger the fault than delta debugging. For some programs, delta debugging can achieve similar reductions than genetic algorithm, but in the genetic algorithm can achieve bigger reductions.

RQ14. How much data reduce the input reduction techniques? The genetic algorithm and delta debugging reduce the majority of the data. However, the genetic algorithm can reduce more data.

RQ15. How much execution time is employed by the input reduction techniques of the MapReduce design faults? Delta debugging is scalable employing few seconds, but the genetic algorithm does not scale rightly and employs minutes-hours. The delta debugging technique converges in the solution faster than genetic algorithm, but the genetic algorithm also searches more exhaustively a better solution that sometimes achieves.

The testing technique is able to detect the design fault and the debugging framework provides the potential causes of this fault to make it easy to understand and fix. These techniques can also be used to detect faults in the programs executed in production. The **fourth line of research** is focused in the design fault during the operations (Section V). The programs executed in production can be tested automatically through the testing technique proposed in the thesis and taking samples of runtime data as input. Once the fault is detected, the root cause of the fault can also be automatically obtained through the debugging techniques proposed in the thesis.

VI.2 Conclusions

There are several challenges to test the functionality of the *MapReduce* programs due the execution over large-scale infrastructure with frequent infrastructure failures. Some of the functional faults may be masked in a testing/development environment but revealed in production because the program is wrongly designed to support optimizations and concurrency at large scale.

These faults can be detected executing the test cases in different configurations that reproduce the production environment with the non-deterministic executions due the frequent infrastructure failures and optimizations, among others. We devised a testing technique and a tool that checks that the same input executed in different configurations provides similar outputs, in other case a design fault is detected. Usually this technique detects the design faults automatically in few seconds.

However, these design faults are not easy to debug due the parallel execution of the code and the internal optimizations. However, the root cause of the fault can be automatically localized analysing those design characteristics that are usually more prone to trigger the failure and those that masked the fault. We devised a fault localization technique and tool that (1) generate several relevant configurations, and (2) analyse statistically the characteristics of these configurations that are more prone to fail and succeed. This technique provides automatically in few seconds a rank of the more suspicious characteristics of the design, and the root cause of the design fault is usually the top.

Although the root cause of the fault is localized, these design faults can not be easy to understand because the test cases fail in executions that involves concurrency and optimizations. We devised an input reduction technique to isolate the relevant data that still trigger the failure making the fault easier to understand. This technique reduces the majority of the input data in few seconds. The input reduction and fault localization techniques are integrated in a debugging framework that also support the inspection of the code though the common debugging utilities such as breakpoints and watchpoints.

The previous testing and debugging techniques can be used automatically in laboratory, but they can also be used in production with the runtime programs. We devised an autonomous environment to detect design faults in the *MapReduce* programs executed in production taking samples of the runtime data as input of test cases. Once a design failure is detected automatically in few seconds of execution, the developed is informed about the fault and its root cause. Then the developer can stop the *Big Data* program saving money, energy, computer resources and potential failures.

VI.3 Future Work

The techniques proposed in the thesis are able to detect and debug the design faults of the *MapReduce* programs automatically in both laboratory and production. As future work we plan to also fix the *MapReduce* programs automatically through self-adaptation technique. Some of the design faults that are caused by optimizations can easily fixed turning down the optimizations or with little modifications. In other cases, the faults are difficult to fix like in those with domain-specific semantic in the design. There are also other cases in which the automatic fix is not possible because the whole program functionality does not fit in the *MapReduce* processing model. Currently, we are working in a PDCA methodology to not only fix the program, but also measure the impact of the functional patch in performance and maintenance. We plan to automatize this methodology through a search-based approach.

As future work, we also plan to adapt the testing and debugging techniques to other systems and integrate them with other technologies of the *Big Data* stack. Some frameworks very used in *Big Data* like Spark and Flink extend and generalize the *MapReduce* processing model adding different operations that can also yield in non-deterministic faults. The techniques proposed in this thesis can be slightly expanded to address these new kinds of faults.

VII CONSLUSIONES Y TRABAJO FUTURO

En las pruebas funcionales de los programas *MapReduce* hay varios desafíos debido a la ejecución en una infraestructura de gran escala con frecuentes fallos de infraestructura. Algunos de los defectos funcionales pueden estar enmascarados en un entorno de pruebas/desarrollo pero revelados en producción porque el programa está incorrectamente diseñado para soportar optimización y concurrencias a gran escala.

Estos defectos pueden ser detectados ejecutando los casos de prueba en diferentes configuraciones debido a los frecuentes fallos de infraestructura y optimizaciones, entre otros. Hemos diseñado una técnica de prueba y una herramienta que comprueba que la ejecución de la misma entrada en diferentes configuraciones provea salidas similares, en caso contrario se detecta un defecto de diseño. Normalmente esta técnica detecta los defectos de diseño automáticamente en pocos segundos.

Sin embargo, estos defectos de diseño no son fáciles de depurar por la ejecución paralela del código y las optimizaciones internas. No obstante, la causa raíz del defecto pude ser localizada automáticamente analizando tanto aquellas características de diseño que habitualmente son más propensas a provocar el fallo, como aquellas otras que encascaran el defecto. Hemos diseñado una técnica de localización de defectos y una herramienta que (1) genera varias configuraciones, y (2) analiza estadísticamente las características de esas configuraciones que son más propensas a fallar y a tener éxito. Esta técnica proporciona automáticamente en pocos segundos un ranking de las características de diseño más sospechosas, y la causa raíz del defecto de diseño suele estar en las primeras posiciones.

Aunque se localice la cusa raíz del defecto, estos defectos de diseño no pueden ser fácilmente entendibles porque los casos de prueba fallan en ejecuciones que involucran concurrencia y optimizaciones. Hemos diseñado una técnica de reducción de datos que aísla los datos relevantes que siguen causando el fallo, haciendo que el defecto sea más fácil de entender. Esta técnica reduce la mayoría de los datos de entrada en pocos segundos. Las técnicas de reducción de datos y de localización de defectos se han integrado en un framework de depuración que también soporta la inspección del código a través de las utilidades habituales de la depuración como los breakpoints y los watchpoints.

Las anteriores técnicas de pruebas y depuración pueden ser utilizadas automáticamente en el laboratorio, pero también pueden utilizarse en producción con los programas que se están ejecutando. Hemos diseñado un entorno autónomo que detecta los defectos de diseño de los programas *MapReduce* que se ejecutan en producción tomando como entradas de los casos de pruebas muestras de los datos que se están procesando en producción. Una vez que un fallo de diseño es detectado automáticamente en pocos segundos de ejecución, el desarrollador es informado sobre el defecto y su causa raíz. Entonces el desarrollador puede parar el programa *Big Data* para ahorrar dinero, energía, recursos de computación y evitar potenciales fallos.

Las técnicas propuestas en la tesis son capaces de detectar y depurar automáticamente los defectos de diseño de los programas *MapReduce* tanto en el laboratorio como en producción. Como trabajo futuro planeamos también corregir automáticamente los programas *MapReduce* utilizando una técnica de autoadaptación. Algunos de los defectos de diseño son causados por optimizaciones que pueden ser fácilmente corregibles deshabilitando la optimización o con pequeñas modificaciones. En otros casos, los defectos son difíciles de corregir como ocurre con aquellos que tienen en el diseño semánticas específicas de un dominio. También hay otros casos en los que la corrección automática no es posible porque toda la funcionalidad del programa no es compatible con el modelo de procesamiento *MapReduce*. Actualmente estamos trabajando en una metodología PDCA que no sólo corrija el programa, sino que también mide el impacto que tiene la mejora funcional respecto al rendimiento y el mantenimiento. Planeamos automatizar esta metodología a través de un enfoque guiado por búsqueda.

Como trabajo futuro también planeamos adaptar las técnicas de pruebas y depuración a otros sistemas e integrarlas con otras tecnologías *Big Data*. Algunos frameworks muy utilizados en *Big Data* como Spark y Flink extienden y generalizan el modelo de procesamiento *MapReduce* añadiendo diferentes operaciones que también pueden causar defectos no-determinísticos. Las técnicas propuestas en esta tesis pueden expandirse ligeramente para abordar estos nuevos tipos de defectos.

VIII APPENDICES

VIII.1 HIGH-IMPACT JOURNALS AND CONFERENCES USED FOR THE MAPPING STUDY

Table 23: High-impact journals for the mapping study

	2015				2016			
JCR journals	**Rank**	**Impact factor**	**Number of citations**	**Number of papers**	**Rank**	**Impact factor**	**Number of citations**	**Number of papers**
ACM Computing Surveys	Q1	5.24	4150	88	Q1	6.75	6629	76
ACM SIGPLAN Notices	Q1	0.49	3657	389	Q1	0.34	2541	378
ACM Transactions on Database Systems (ACM TODS)	Q2	0.63	969	19	Q2	1.52	1504	26
ACM Transactions on Information Systems (ACM TOIS)	Q2	0.98	1220	27	Q2	2.31	1790	33
ACM Transactions on Software Engineering and Methodology (ACM TOSEM)	Q3	1.51	700	21	Q2	2.52	1104	16
Computer Science and Information Systems (ComSIS)	Q4	0.62	265	64	Q4	0.84	392	47
Distributed and Parallel Databases	Q4	0.80	293	21	Q4	1.18	349	19
Distributed Computing	Q3	1.26	498	26	Q2	1.67	954	24
Empirical Software Engineering (ESE)	Q2	1.39	828	52	Q2	3.28	1453	68
The International Arab Journal of Information Technology (IAJIT)	Q4	0.52	292	78	Q3	0.72	502	93
IEEE Software	Q2	0.82	1638	55	Q1	2.19	2547	69
IEEE Transactions on Knowledge and Data Engineering (IEEE TKDE)	Q1	2.48	6465	245	Q1	3.44	9370	239
IEEE Transactions on Parallel and Distributed Systems (IEEE TPDS)	Q1	2.66	5080	282	Q1	4.18	8313	271
IEEE Transactions on Software Engineering (IEEE TSE)	Q1	1.51	4221	62	Q1	3.27	6712	59
International Journal of Data Warehousing and Mining (IJDWM)	Q4	0.63	146	17	Q4	0.73	219	15
International Journal of Information Management (IJIM)	Q1	2.69	1937	73	Q1	3.87	3087	115
International Journal of Information Processing and Management (IJIPM)	Q1	1.40	2296	63	Q1	2.39	3067	72
International Journal of Information Technology and Decision Making (IJITDM)	Q3	1.18	627	45	Q3	1.66	742	56

International Journal of Information Technology and Management (IJITM)	Q4	0.60	226	23	Q4	1.07	281	29
International Journal of Software Engineering and Knowledge Engineering (IJSEKE)	Q4	0.24	216	55	Q4	0.30	345	52
Information and Software Technology (IST)	Q1	1.57	2145	153	Q1	2.69	3448	122
Journal of Database Management (JDM)	Q4	0.12	131	7	Q4	0.27	182	9
Journal of Information Technology (JIT)	Q1	4.78	1695	24	Q1	6.95	2515	19
Journal of Management Information Systems (JMIS)	Q1	3.03	3818	41	Q1	2.36	4456	30
Journal of Software: Evolution and Process	Q4	0.73	140	46	Q4	1.03	319	50
Journal of Parallel and Distributed Computing (JPDC)	Q1	1.32	1983	94	Q1	1.93	2740	84
The Journal of Strategic Information Systems (JSIS)	Q2	2.60	1159	17	Q2	3.49	1580	15
Journal of Systems and Software (JSS)	Q1	1.42	3243	181	Q1	2.44	5161	229
Knowledge and Information Systems (KAIS)	Q2	1.70	1559	110	Q2	2.00	2146	117
Software Quality Journal (SQJ)	Q4	0.79	280	24	Q3	1.86	486	33
Software Testing, Verification& Reliability (STVR)	Q3	1.08	363	25	Q3	1.59	612	20

Table 24: CORE conferences for the mapping study

CORE Conferences	**CORE 2014**	**CORE 2017**
ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)	A*	A*
Computer Aided Verification (CAV)	A*	A*
IEEE International Conference on Data Mining (IEEE ICDM)	A*	A*
International Conference on Data Engineering (IEEE ICDE)	A*	A*
Special Interest Group on Management of Data Conference (SIGMOD)	A*	A*
Very Large Data Bases Conference (VLDB)	A*	A*
Automated Software Engineering (ASE)	A	A
Biennial Conference on Innovative Data Systems Research (CIDR)	A	A
Empirical Software Engineering and Measurement (ESEM)	A	A
European Conference on Parallel Processing (EURO-PAR)	A	A
European Conference on Principles of Data Mining and Knowledge Discovery (PKDD)	A	A

International Conference on Database Theory (ICDT)	A	A
International Conference on Distributed Computing Systems (ICDCS)	A	A
International Conference on Extending Database Technology (EDBT)	A	A
International Conference on Information and Knowledge Management (CIKM)	A	A
International Conference on Software Engineering (ICSE)	A	A
International Conference on Statistical and Scientific Database Management (SSDBM)	A	A
International Symposium on Cluster Computing and the Grid (CCGRID)	A	A
International Symposium on Intelligent Data Analysis (IDA)	A	A
International Symposium on Software Testing and Analysis (ISSTA)	A	A
Joint International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)	A	A
Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)	A	A
Parallel Computing Technologies International Conferences Series (PaCT)	A	A
SIAM International Conference on Data Mining (SDM)	A	A
Symposium on Large Spatial Databases (SSTD)	A	A
ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)	A	B
Advances in Databases and Information Systems (ADBIS)	B	B
Australasian Data Mining Conference (AusDM)	B	B
Australasian Database Conference (ADC)	B	B
Databases and Programming Language (DBPL)	B	B
European Software Engineering Conference (ESEC)	A	B
IEEE International Conference on Cloud Computing (IEEE CLOUD)	B	B
IEEE International Enterprise Distributed Object Computing Conference (IEEE EDOC)	B	B
International Baltic Conference on Databases and Information Systems (DB&IS)	B	B
International Conference on Data Warehousing and Knowledge Discovery (DaWaK)	B	B
International Conference on Database and Expert Systems Applications (DEXA)	B	B
International Conference on Database Systems for Advanced Applications (DASFAA)	B	B
International Conference on Management of Data (COMAD)	B	B
International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)	B	B
International Conference on Quality Software (QSIC)	B	B
International Conference on Software and Data Technologies (ICSOFT)	B	B
International Conference on Tests and Proof (TAP)	B	B
International Database Engineering and Applications Symposium (IDEAS)	B	B
International Workshop on Data Warehousing and OLAP (DOLAP)	B	B

Software Engineering and Knowledge Engineering (SEKE)	B	B
Symposium on Applied Computing (SAC)	B	B
Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)	C	C
Evolution and Change in Data Management (ECDM)	C	C
IEEE International Conference on Cloud Computing Technology and Science (IEEE CloudCom)	C	C
International Conference on Intelligent Data Engineering and Automated Learning (IDEAL)	C	C
International Conference on Software Testing, Verification and Validation (ICST)	C	C
International Workshop on Formal Approaches to Testing of Software (FATES)	C	C
Symposium on Principles of Database Systems (PODS)	C	C

VIII.2 OTHER JOURNALS AND CONFERENCES USED FOR THE MAPPING STUDY

Table 25: Other journals for the mapping study

ACM DATA BASE	International Journal of Intelligent Information and Database Systems (IJIIDS)
ACM SIGSOFT Software Engineering Notes (ACM SIGSOFT)	International Journal of Information Quality (IJIQ)
ACM Transactions on Management Information Systems (ACM TMIS)	International Journal of Information Systems and Change Management (IJISCM)
Big Data Research	International Journal of Information Technologies and Systems Approach (IJITSA)
Computing and Information Technology (CIT)	International Journal of Parallel, Emergent and Distributed Systems (IJPEDS)
European Journal of Information Systems (EJIS)	Journal of Cases on Information Technology (JCIT)
Foundations and Trends in Databases (FTDB)	Journal of Data and Information Quality (JDIQ)
IEEE Cloud Computing	Journal of Digital Information Management (JDIM)
IEEE Computer	Journal of Enterprise Information Management (JEIM)
IEEE Distributed Systems Online (IEEE DS)	Journal of Information and Data Management (JIDM)
IEEE Transactions on Big Data	Journal of Information
IEEE Transactions on Cloud Computing (IEEE TCC)	Journal of Information Processing (JIP)
International Journal of Big Data Intelligence (IJBD)	The Journal of Information Processing Systems (JIPS)
International Journal of Cloud Applications and Computing (IJCAC)	Journal of Information Technology Research (JITR)

International Journal of Cloud Computing (IJCC)	Journal of Systems and Information Technology (JSIT)
International Journal of Distributed Systems and Technologies (IJDST)	Transactions on Large-Scale Data- and Knowledge-Centered Systems (Transactions LDKS)
International Journal of Enterprise Information Systems (IJEIS)	Journal of Enterprise Information Management (JEIM)

Table 26: Other conferences for the mapping study

Advances in Model-Based Testing (A-MOST)	Industrial Conference on Data Mining (ICDM)
Alberto Mendelzon Workshop on Foundations of Data Management (AMW)	International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS)
International Conference on Big Data Analytics (BDA)	Internet and Distributed Computing Systems (IDCS)
International Conference Beyond Databases, Architectures, and Structures (BDAS)	IEEE/ACM International Symposium on Big Data Computing (BDC)
International Conference on Big Data and Smart Computing (BigComp)	IEEE International Conference on Big Data (IEEE BigData)
International Congress on Big Data (BigData Congress)	IEEE Symposium on Large-Scale Data Analysis and Visualization (IEEE LDAV)
Workshop on Scalability in Model Driven Engineering (BigMDE)	International Conference on Algorithms for Big Data (ICABD)
British National Conference on Databases (BNCOD)	International Conference on Big Data and Cloud Computing (BdCloud)
International Conference on Cloud and Autonomic Computing Conference (CAC)	International Conference on Big Data Cloud and Applications (BDCA)
International Conference on Cloud and Green Computing (CGC)	International Conference on Big Data Computing and Communications (BigCom)
International Conference on Cloud Computing and Services Science (CLOSER)	International Conference on Big Data Computing Service and Applications (BigDataService)
Cloud Computing (CloudComp)	International Multiconference on Computer Science and Information Technology (IMCSIT)
Conference on Data and Application Security and Privacy (CODASPY)	International Workshop on Machine Learning, Optimization, and Big Data (MOD)
International Computer Software and Applications Conference (COMPSAC)	Symposium on Network Cloud Computing and Applications (NCCA)
International Conference on Cloud and Service Computing (CSC)	Conference on Next Generation Information Technologies and Systems (NGITS)
European Joint Conference on Theory and Practice of Software (ETAPS)	ACM Symposium on Cloud Computing (SoCC)
International Conference on Future Data and Security Engineering (FDSE)	SPIN Workshop on Model Checking of Software (SPIN)

Federated Conference on Computer Science and Information Systems (FEDCSIS)	Symposium on Computational Intelligence in Big Data (CIBD)
International Conference on Future Internet of Things and Cloud (FICLOUD)	Symposium on Information Management and Big Data (SIMBig)
USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)	Testing: Academic
International Conference on Advanced Cloud and Big Data (CBD)	International Conference on Testing Communicating Systems (TestCom)
International Conference on Cloud Engineering (IC2E)	Workshop on Big Data Benchmarking (WBDB)
International Conference on Algorithms for Big Data (ICABD)	Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware (BPOE)
International Conference on Innovative Computing and Cloud Computing (ICCC)	Workshop on Mobile Big Data (Mobidata)
International Conference on Data Engineering and Management (ICDEM)	

VIII.3 Primary Studies

Table 27: Primary studies

Ref.	Year	Contribution	Number of citations[1]	Summary
[142]	2013	Conference	20	A study and characterization of MapReduce-like failures
[112]	2013	Conference	30	A prediction model of individual MapReduce jobs based on important properties
[120]	2013	Conference	25	A performance prediction based on network properties and configuration of the cluster
[138]	2013	Conference	22	A performance prediction based on a representation of the architecture with some information of the MapReduce program
[150]	2015	Conference	7	Generator of representative data to testing Big Data programs based on input space partitioning
[27]	2010	Conference	311	A study and characterization of more than 170000 MapReduce executions
[114]	2011	Conference	46	A simple performance prediction model that considers the program and the system
[119]	2013	Journal	47	A model that obtains several metrics about the MapReduce programs performance and resource utilization
[109]	2013	Briefing	19	Classification of testing in Big Data and the underlying challenges
[143]	2013	Conference	8	Classification of MapReduce faults based on empirical changes in the programs

[144]	2015	Conference	13	Checking of the commutativity problem in the Reduce functions
[121]	2013	Conference	30	A performance prediction model based on information about the MapReduce program and the cluster
[136]	2012	Doctoral dissertation	14	A simulator of MapReduce program that obtains a prediction of the performance
[117]	2014	Journal	50	A performance prediction model of MapReduce program using Mean Field Analysis and information of the program, system and data
[127]	2015	Conference	7	A performance prediction model of MapReduce program in the cloud considering the program and the data
[148]	2012	Conference	34	A failure injector in the architecture using the cloud manager in order to test the MapReduce programs
[110]	2013	Conference	7	Challenges of software testing in Big Data
[28]	2013	Conference	97	A study and characterization of three Hadoop clusters
[129]	2016	Journal	31	Prediction of the performance and optimization of resource utilization based on deadline requirements
[139]	2011	Conference	54	Monitoring of the MapReduce program that generates detailed reports of the execution
[141]	2013	Journal	54	A study and characterization of several bugs in Big Data programs
[128]	2015	Conference	10	A performance prediction model of the MapReduce programs considering the deployment in virtualized cloud and the characteristics of the program
[118]	2012	Conference	11	A performance prediction model of the MapReduce programs considering several samplings of the input data
[147]	2015	Journal	11	A testing framework to run the MapReduce programs under architectural failures in order to test
[122]	2013	Conference	7	A performance prediction model of the MapReduce programs considering the resource contention and the task failures
[38]	2014	Conference	7	Classification of several MapReduce faults with a series of challenges in order to reveal the faults
[145]	2011	Conference	28	Functional Testing of the Reduce function based on symbolic execution
[29]	2014	Conference	22	Characterization of the MapReduce programs based on empirical study

[123]	2014	Conference	5	A performance prediction model of the MapReduce programs considering the memory shared and disk I/O
[103]	2014	Journal	9	Prediction of the MapReduce performance based on empirical executions and an adjustment based on micro benchmarks
[140]	2014	Journal	19	Performance analysis model for MapReduce applications based on ISO 25010 that establishes a relationship between the performance and reliability measures
[115]	2013	Conference	1	Obtains the performance of the MapReduce programs based on Stochastic Petri Nets
[133]	2015	Conference	2	Performance prediction of HIVE-QL queries through the underlying MapReduce applications based on multiple lineal regression
[105]	2013	Journal	13	Performance prediction of PIG queries through the underlying MapReduce applications
[113]	2014	Conference	2	Performance prediction for a MapReduce program and optimization based on the type of application and potential bottlenecks
[124]	2014	Conference	6	Mathematical model for performance prediction of the RDMA-Enhanced MapReduce programs
[104]	2013	Conference	32	A performance prediction model of the MapReduce programs considering information of the program and the performance for several parts of the program
[130]	2015	Conference	3	Model that predicts the performance of MapReduce applications in hybrid clouds
[125]	2015	Conference	33	Simulation of the Spark applications in order to obtain performance information
[126]	2014	Conference	1	A performance prediction model of the MapReduce programs considering the heterogeneity of the cluster
[116]	2011	Conference	29	A performance prediction model of the MapReduce programs based on the mean time between failures
[111]	2014	Conference	8	Overview and challenges of performance testing in Big Data
[151]	2013	Conference	8	Data generation for dataflow programs based on symbolic execution
[135]	2014	Conference	8	Simulating the MapReduce program under configurable hardware in order to obtain a performance prediction
[137]	2014	Conference	2	Simulating the scheduler of the MapReduce program in order to test the best configuration
[107]	2015	Conference	6	Test factory model for Big Data development

[153]	2011	Conference	71	Static analysis of the MapReduce configuration in order to detect misconfigurations and avoid failures
[152]	2011	Conference	12	Automatic checking of the java types inside MapReduce programs in order to detect incompatible types
[149]	2012	Dissertation	7	Data generator for MapReduce programs based on bacteriological algorithm in order to test the program
[53]	2015	Conference	6	Testing technique for MapReduce programs based on data flow and the MapReduce specifics
[146]	2013	Conference	5	Checking the correctness of the dataflow programs based on the operators properties
[134]	2013	Journal	0	Performance prediction of the join queries in Pig
[108]	2013	Journal	11	Overview and challenges of testing in Big Data
[154]	2011	Conference	17	Formal verification of the MapReduce program based on a model of the program/specification and invariants

[1] Number of citations obtained from Google Scholar [276] in 2018.

REFERENCES

[1] J. Gantz, D. Reinsel, and B. D. Shadows, “The Digital Universe in 2020,” *IDC iView “Big Data, Bigger Digit. Shad. Biggest Growth Far East,”* vol. 2007, no. December 2012, pp. 1–16, 2012.

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

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

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

[5] McKinsey & Company, “Big data: The next frontier for innovation, competition, and productivity,” *McKinsey Glob. Inst.*, no. June, p. 156, 2011.

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

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

[8] B. Marr, “Where Big Data Projects Fail,” 2015. [Online]. Available: http://www.forbes.com/sites/bernardmarr/2015/03/17/where-big-data-projects-fail/. [Accessed: 19-Feb-2017].

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

[10] N. Laranjeiro, S. N. Soydemir, and J. Bernardino, “A Survey on Data Quality: Classifying Poor Data,” in *Proceedings - 2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing, PRDC 2015*, 2016, pp. 179–188.

[11] 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.

[12] IBM, “The Four V’s of Big Data,” 2013.

[13] United States Postal Service, “Undeliverable as Addressed Mail,” 2014.

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

[15] 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.

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

[17] “Institutions that are using Apache Hadoop for educational or production uses.” [Online]. Available: https://wiki.apache.org/hadoop/PoweredBy. [Accessed: 23-Jan-2017].

[18] S. Agarwal and Z. Khanam, “Map Reduce: A Survey Paper on Recent Expansion,” *Int. J. Adv. Comput. Sci. Appl.*, vol. 6, no. 8, 2015.

[19] Z. Khanam and S. Agarwal, “Map-Reduce Implementations: Survey and Performance Comparison,” *Int. J. Comput. Sci. Inf. Technol.*, vol. 7, no. 4, pp. 119–126, 2015.

[20] “Apache Hadoop: open-source software for reliable, scalable, distributed computing.”

[Online]. Available: https://hadoop.apache.org/. [Accessed: 23-Jan-2017].

[21] "Apache Flink: Scalable batch and stream data processing." [Online]. Available: https://flink.apache.org. [Accessed: 20-Feb-2017].

[22] A. Alexandrov, R. Bergmann, S. Ewen, J. C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The Stratosphere platform for big data analytics," *VLDB J.*, vol. 23, no. 6, pp. 939–964, 2014.

[23] Apache Spark, "Apache Spark: a fast and general engine for large-scale data processing," *Spark.Apache.Org*, 2015. [Online]. Available: https://spark.apache.org. [Accessed: 20-Feb-2017].

[24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster Computing with Working Sets," *HotCloud'10 Proc. 2nd USENIX Conf. Hot Top. cloud Comput.*, p. 10, 2010.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] ISO/IEC/IEEE, "29119-1:2013 - ISO/IEC/IEEE International Standard for Software and systems engineering — Software testing — Part 1: Concepts and definitions," *ISO/IEC/IEEE 29119-1:2013(E)*, vol. 2013, pp. 1–64, 2013.

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

[32] A. Bertolino, P. Inverardi, and H. Muccini, "Software architecture-based analysis and testing: A look into achievements and future challenges," *Computing*, vol. 95, no. 8. pp. 633–648, 2013.

[33] B. A. Kitchenham, D. (David) Budgen, and P. Brereton, *Evidence-based software engineering and systematic reviews*. CRC Press, 2015.

[34] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," *EASE'08 Proc. 12th Int. Conf. Eval. Assess. Softw. Eng.*, pp. 68–77, 2008.

[35] D. E. Avison, F. Lau, M. D. Myers, and P. A. Nielsen, "Action research," *Commun. ACM*, vol. 42, no. 1, pp. 94–97, 1999.

[36] N. Juristo and A. M. Moreno, “Basics of Software Engineering Experimentation,” *Analysis*, vol. 5/6, p. 420, 2001.

[37] J. Morán, C. De La Riva, and J. Tuya, “Testing MapReduce Programs: A Systematic Mapping Study,” *J. Softw. Evol. Process*, vol. 31, no. 3, 2019.

[38] 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.

[39] J. Morán, C. De La Riva, and J. Tuya, “Pruebas funcionales en programas MapReduce basadas en comportamientos no esperados,” in *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2014.

[40] J. Morán and C. De La Riva, “Pruebas para sistemas con procesamiento y transformación de datos en paralelo,” University of Oviedo, 2014.

[41] J. Moran, A. Bertolino, C. de la Riva, and J. Tuya, “Automatic Testing of Design Faults in MapReduce Applications,” *IEEE Transactions on Reliability*, 2018.

[42] 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.

[43] 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.

[44] J. Morán, C. De La Riva, and J. Tuya, “Generación y Ejecución de Escenarios de Prueba para Aplicaciones MapReduce,” in *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2016.

[45] T. Chen, S. Cheung, and S. Yiu, “Metamorphic testing: a new approach for generating next test cases,” *Tech. Rep. HKUST-CS98-01, Dep. Comput. Sci. Hong Kong Univ. Sci. Technol. Hong Kon*, pp. 1–11, 1998.

[46] J. Moran, C. De La Riva, and J. Tuya, “Automatización de la localización de defectos en el diseño de aplicaciones MapReduce,” in *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2018.

[47] J. Morán, C. De La Riva, J. Tuya, and B. Rivas, “Localización de defectos en aplicaciones MapReduce,” in *Jornadas de Ingeniería del Software y Bases de Datos*, 2017.

[48] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, 2002.

[49] A. Zeller, “Yesterday, my program worked. Today, it does not. Why?,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 253–267, 1999.

[50] J. H. Holland, *Adaptation in Natural and Artificial Systems*. 1975.

[51] D. E. Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning*. 1989.

[52] J. Moran, A. Bertolino, C. de la Riva, and J. Tuya, “Towards Ex Vivo Testing of MapReduce Applications,” in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2017, pp. 73–80.

[53] 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.

[54] J. Morán, C. De La Riva, and J. Tuya, "Pruebas basadas en flujo de datos para programas MapReduce," in *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2015.

[55] S. Rapps and E. J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 4, pp. 367–375, 1985.

[56] P. Ferrera, I. De Prado, E. Palacios, J. L. Fernandez-Marquez, and G. Di Marzo Serugendo, "Tuple MapReduce and Pangool: an associated implementation," *Knowl. Inf. Syst.*, vol. 41, no. 2, pp. 531–557, 2014.

[57] J. Lin, "Mapreduce is Good Enough? If All You Have is a Hammer, Throw Away Everything That's Not a Nail!," *Big Data*, vol. 1, no. 1, pp. 28–37, 2013.

[58] S. Babu, "Towards automatic optimization of MapReduce programs," *SoCC '10 Proc. 1st ACM Symp. Cloud Comput.*, pp. 137–142, 2010.

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

[60] "JUnit: a simple framework to write repeatable tests." [Online]. Available: http://junit.org. [Accessed: 27-Feb-2017].

[61] "Apache MRUnit: Java library that helps developers unit test Apache Hadoop map reduce job." [Online]. Available: http://mrunit.apache.org. [Accessed: 23-Jan-2017].

[62] "Minicluster: Apache hadoop cluster in memory for testing." [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/CLIMiniCluster.html. [Accessed: 27-Feb-2017].

[63] "Herriot: Large-scale automated test framework." [Online]. Available: https://wiki.apache.org/hadoop/HowToUseSystemTestFramework. [Accessed: 27-Feb-2017].

[64] "Anarchy Ape: Fault injection tool for Hadoop cluster from Yahoo anarchyape." [Online]. Available: https://github.com/david78k/anarchyape. [Accessed: 23-Jan-2017].

[65] "Chaos Monkey: Fault injector." [Online]. Available: https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey. [Accessed: 23-Jan-2017].

[66] "Hadoop Injection Framework." [Online]. Available: https://hadoop.apache.org/. [Accessed: 23-Jan-2017].

[67] B. Kitchenham and S. Charters, "Guidelines for performing Systematic Literature Reviews in Software Engineering," *Engineering*, vol. 2, p. 1051, 2007.

[68] Z. Pan and G. Kosicki, "Framing analysis: An approach to news discourse," *Polit. Commun.*, vol. 10, no. 1, pp. 55–75, 1993.

[69] G. Hart, "The Five W's: An Old Tool for the New Task of Audience Analysis," *Tech. Commun.*, vol. 43, no. 2, pp. 139–145, 1996.

[70] R. Kipling, *Just so stories*. Macmillan and Co, 1902.

[71] C. Jia, Y. Cai, Y. T. Yu, and T. H. Tse, "5W+1H pattern: A perspective of systematic mapping studies and a case study on cloud software testing," *J. Syst. Softw.*, vol. 116, pp. 206–219, 2016.

[72] International Organization For Standardization Iso, “Iso/Iec 25010:2011,” *Softw. Process Improv. Pract.*, vol. 2, no. Resolution 937, pp. 1–25, 2011.

[73] International Organization For Standardization Iso, “ISO/IEC 9126-1,” *Software Process: Improvement and Practice*, vol. 2, no. 1. pp. 1–25, 2001.

[74] “DBLP - digital bibliography & library project.” [Online]. Available: http://dblp.uni-trier.de. [Accessed: 27-Feb-2017].

[75] “Thomson reuters.” [Online]. Available: http://thomsonreuters.com. [Accessed: 27-Feb-2017].

[76] “CORE - Computing research and education.” [Online]. Available: http://www.core.edu.au. [Accessed: 27-Feb-2017].

[77] “IEEE Xplore Digital Library.” [Online]. Available: http://ieeexplore.ieee.org. [Accessed: 27-Feb-2017].

[78] “ACM Digital Library.” [Online]. Available: http://dl.acm.org. [Accessed: 27-Feb-2017].

[79] “Scopus.” [Online]. Available: http://www.scopus.com. [Accessed: 27-Feb-2017].

[80] “EI compendex.” .

[81] “ISI web of science.” [Online]. Available: https://www.accesowok.fecyt.es. [Accessed: 27-Feb-2017].

[82] M. Palacios, J. García-Fanjul, and J. Tuya, “Testing in Service Oriented Architectures with dynamic binding: A mapping study,” *Inf. Softw. Technol.*, vol. 53, no. 3, pp. 171–189, 2011.

[83] B. Kitchenham and P. Brereton, “A systematic review of systematic review process research in software engineering,” *Inf. Softw. Technol.*, vol. 55, no. 12, pp. 2049–2075, 2013.

[84] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, “Ganesha: BlackBox Diagnosis of MapReduce Systems,” *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 3, pp. 8–13, 2010.

[85] 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.

[86] S. Tilley and T. Parveen, “HadoopUnit: Test Execution in the Cloud,” in *Software Testing in the Cloud*, 2012, pp. 37–53.

[87] I. Saleh and K. Nagi, “HadoopMutator: A Cloud-Based Mutation Testing Framework,” *14th Int. Conf. Softw. Reuse, ICSR 2015*, pp. 172–187, 2014.

[88] ISO/IEC/IEEE, “29119-4:2015 - ISO/IEC/IEEE International Standard for Software and systems engineering — Software testing — Part 4: Test techniques,” *ISO/IEC/IEEE 29119-4:2015*, pp. 1–149, 2015.

[89] “ISO/IEC 12207:2008 Systems and software engineering - Software life cycle processes.” 2008.

[90] International Software Testing Qualifications Board (ISTQB), “Foundation Level Syllabus Reference, International Software Testing Qualifications Board (ISTQB) Std.” 2011.

[91] M. Shaw, “Writing Good Software Engineering Research Papers,” in *Proceedings of the 25th International Conference on Software Engineering*, 2003, pp. 726–736.

[92] D. S. Cruzes and T. Dyb, “Research synthesis in software engineering: A tertiary study,” in *Information and Software Technology*, 2011, vol. 53, no. 5, pp. 440–455.

[93] D. S. Cruzes and T. Dyba, “Recommended Steps for Thematic Synthesis in Software Engineering,” *2011 Int. Symp. Empir. Softw. Eng. Meas.*, no. 7491, pp. 275–284, 2011.

[94] G. W. Noblit and R. D. R. Hare, *Meta-ethnography: Synthesizing qualitative studies*, vol. 11. 1988.

[95] A. Strauss and J. Corbin, “Basics of qualitative research: grounded theory procedure and techniques,” *Qual. Sociol.*, vol. 13, no. 1, pp. 3–21, 1990.

[96] F. Q. B. Da Silva, S. S. J. O. Cruz, T. B. Gouveia, and L. F. Capretz, “Using meta-ethnography to synthesize research: A worked example of the relations between personality and software team processes,” in *International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 153–162.

[97] “ACM SIGSOFT - SEWORLD Mailing List.” [Online]. Available: https://www.sigsoft.org/resources/seworld.html. [Accessed: 29-Mar-2018].

[98] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, “Lessons from applying the systematic literature review process within the software engineering domain,” *J. Syst. Softw.*, vol. 80, no. 4, pp. 571–583, 2007.

[99] M. Petticrew and H. Roberts, *Systematic reviews in the social sciences : a practical guide*. John Wiley & Sons, 2008.

[100] K. Petersen, S. Vakkalanka, and L. Kuzniarz, “Guidelines for conducting systematic mapping studies in software engineering: An update,” in *Information and Software Technology*, 2015, vol. 64, pp. 1–18.

[101] J. Cohen, “A Coefficient of Agreement for Nominal Scales,” *Educ. Psychol. Meas.*, vol. 20, no. 1, pp. 37–46, 1960.

[102] M. L. McHugh, “Interrater reliability: the kappa statistic,” *Biochem. Medica*, pp. 276–282, 2012.

[103] Z. Zhang, L. Cherkasova, and B. T. Loo, “Parameterizable benchmarking framework for designing a mapreduce performance model,” *Concurr. Comput. Pract. Exp.*, vol. 26, no. 12, pp. 2005–2026, 2014.

[104] Z. Zhang, L. Cherkasova, and B. T. Loo, “Performance Modeling of MapReduce Jobs in Heterogeneous Cloud Environments,” *Cloud Comput. (CLOUD), 2013 IEEE Sixth Int. Conf.*, pp. 839–846, 2013.

[105] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo, “Performance Modeling and Optimization of Deadline-Driven Pig Programs,” *ACM Trans. Auton. Adapt. Syst.*, vol. 8, no. 3, pp. 1–28, 2013.

[106] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, “Systematic literature reviews in software engineering – A systematic literature review,” *Inf. Softw. Technol.*, vol. 51, pp. 7–15, 2008.

[107] M. Thangaraj and S. Anuradha, “State of art in testing for big data,” in *2015 IEEE International Conference on Computational Intelligence and Computing Research, ICCIC 2015*, 2016.

[108] A. Mittal, “Trustworthiness of Big Data,” *Int. J. Comput. Appl.*, vol. 80, no. 9, pp. 35–40,

Oct. 2013.

[109] 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.

[110] 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.

[111] 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.

[112] G. Song, Z. Meng, F. Huet, F. Magoules, L. Yu, and X. Lin, “A Hadoop MapReduce performance prediction method,” in *Proceedings - 2013 IEEE International Conference on High Performance Computing and Communications, HPCC 2013 and 2013 IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2013*, 2014, pp. 820–825.

[113] J. Yin and Y. Qiao, “Performance modeling and optimization of MapReduce programs,” in *CCIS 2014 - Proceedings of 2014 IEEE 3rd International Conference on Cloud Computing and Intelligence Systems*, 2014, pp. 180–186.

[114] X. Yang and J. Sun, “An analytical performance model of MapReduce,” *2011 IEEE Int. Conf. Cloud Comput. Intell. Syst.*, pp. 306–310, 2011.

[115] S.-T. Cheng, H.-C. Wang, Y.-J. Chen, and C.-F. Chen, “Performance Analysis Using Petri Net Based MapReduce Model in Heterogeneous Clusters,” in *Advances in Web-Based Learning – ICWL 2013 Workshops*, 2015, pp. 170–179.

[116] H. Jin, K. Qiao, X. H. Sun, and Y. Li, “Performance under failures of mapReduce applications,” in *Proceedings - 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2011*, 2011, pp. 608–609.

[117] A. Castiglione, M. Gribaudo, M. Iacono, and F. Palmieri, “Exploiting mean field analysis to model performances of big data architectures,” *Futur. Gener. Comput. Syst.*, vol. 37, pp. 203–211, 2014.

[118] L. Xu, “MapReduce Framework Optimization via Performance Modeling,” in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012, pp. 2506–2509.

[119] E. Vianna, G. Comarela, T. Pontes, J. Almeida, V. Almeida, K. Wilkinson, H. Kuno, and U. Dayal, “Analytical performance models for mapreduce workloads,” *Int. J. Parallel Program.*, vol. 41, no. 4, pp. 495–525, 2013.

[120] J. Han, M. Ishii, and H. Makino, “A Hadoop performance model for multi-rack clusters,” in *2013 5th International Conference on Computer Science and Information Technology, CSIT 2013 - Proceedings*, 2013, pp. 265–274.

[121] M. Ishii, J. Han, and H. Makino, “Design and performance evaluation for Hadoop clusters on virtualized environment,” in *International Conference on Information Networking*, 2013, pp. 244–249.

[122] X. Cui, X. Lin, C. Hu, R. Zhang, and C. Wang, “Modeling the Performance of MapReduce under Resource Contentions and Task Failures,” *2013 IEEE 5th Int. Conf. Cloud Comput. Technol. Sci.*, no. 1, pp. 158–163, 2013.

[123] S. T. Ahmed and D. Loguinov, “On the performance of MapReduce: A stochastic approach,” in *2014 IEEE International Conference on Big Data (Big Data)*, 2014, pp. 49–54.

[124] M. Wasi-ur-Rahman, X. Lu, N. S. Islam, and D. K. Panda, “Performance Modeling for RDMA-Enhanced Hadoop MapReduce,” *2014 43rd Int. Conf. Parallel Process.*, vol. 2014–Novem, no. November, pp. 50–59, 2014.

[125] K. Wang and M. M. H. Khan, “Performance prediction for apache spark platform,” in *Proceedings - 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security and 2015 IEEE 12th International Conference on Embedded Software and Systems, H*, 2015, pp. 166–173.

[126] Y. Fan, W. Wu, Y. Xu, Y. Cao, Q. Li, J. Cui, and Z. Duan, “Performance prediction model in heterogeneous MapReduce environments,” in *Proceedings - 2014 IEEE International Conference on Computer and Information Technology, CIT 2014*, 2014, pp. 240–245.

[127] X. Wu, Y. Liu, and I. Gorton, “Exploring Performance Models of Hadoop Applications on Cloud Architecture,” in *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures - QoSA ’15*, 2015, pp. 93–101.

[128] I. Mytilinis, D. Tsoumakos, V. Kantere, A. Nanos, and N. Koziris, “I/O Performance Modeling for Big Data Applications over Cloud Infrastructures,” in *2015 IEEE International Conference on Cloud Engineering*, 2015, pp. 201–206.

[129] M. Khan, Y. Jin, M. Li, Y. Xiang, and C. Jiang, “Hadoop Performance Modeling for Job Estimation and Resource Provisioning,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 2, pp. 441–454, 2016.

[130] H. Ohnaga, K. Aida, and O. Abdul-Rahman, “Performance of Hadoop Application on Hybrid Cloud,” in *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, 2015, pp. 130–138.

[131] “Apache Hive: data warehouse software facilitates reading, writing, and managing large datasets residing in distributed storage using SQL.” [Online]. Available: https://hive.apache.org/. [Accessed: 25-Oct-2017].

[132] “Apache Pig: platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs.” [Online]. Available: https://pig.apache.org/. [Accessed: 25-Oct-2017].

[133] A. Sangroya and R. Singhal, “Performance Assurance Model for HiveQL on Large Data Volume,” in *Proceedings - 22nd IEEE International Conference on High Performance Computing Workshops, HiPCW 2015*, 2016, pp. 26–33.

[134] R. J. M. Mogrovejo, J. M. Monteiro, J. C. Machado, C. J. M. Viana, and S. Lifschitz, “Towards a Statistical Evaluation of PigLatin Joins,” *J. Inf. Data Manag.*, vol. 4, no. 3, p. 483, 2013.

[135] Z. Bian, K. Wang, Z. Wang, G. Munce, I. Cremer, W. Zhou, Q. Chen, and G. Xu, “Simulating Big Data Clusters for System Planning, Evaluation, and Optimization,” *2014 43rd Int. Conf. Parallel Process.*, pp. 391–400, 2014.

[136] G. Wang, “Evaluating Mapreduce system performance: A Simulation approach,” Virginia Polytechnic Institute and State University, 2012.

[137] J. Chauhan, D. Makaroff, and W. Grassmann, “Simulation and performance evaluation of the hadoop capacity scheduler,” in *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, 2014, pp. 163–177.

[138] E. Barbierato, M. Gribaudo, and M. Iacono, “A performance modeling language for big data architectures,” *Proc. High …*, 2013.

[139] J. Dai, J. Huang, S. Huang, B. Huang, and Y. Liu, “HiTune: dataflow-based performance analysis for big data cloud,” *Proceeding USENIXATC’11 Proc. 2011 USENIX Conf. USENIX Annu. Tech. Conf.*, p. 7, 2011.

[140] L. Bautista Villalpando, A. April, and A. Abran, “Performance analysis model for big data applications in cloud computing,” *J. Cloud Comput. Adv. Syst. Appl.*, vol. 3, no. 1, pp. 19–38, 2014.

[141] A. Rabkin and R. H. Katz, “How hadoop clusters break,” *IEEE Softw.*, vol. 30, no. 4, pp. 88–94, 2013.

[142] S. Li, H. Zhou, H. Lin, T. Xiao, H. Lin, W. Lin, and T. Xie, “A characteristic study on failures of production distributed data-parallel programs,” in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 963–972.

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

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

[145] 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.

[146] Z. Xu, M. Hirzel, G. Rothermel, and K. L. Wu, “Testing properties of dataflow program operators,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, 2013, pp. 103–113.

[147] 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.

[148] 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.

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

[150] N. Li, A. Escalona, Y. Guo, and J. Offutt, “A Scalable Big Data Test Framework,” in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–2.

[151] K. Li, C. Reichenbach, Y. Smaragdakis, Y. Diao, and C. Csallner, “SEDGE: Symbolic example data generation for dataflow programs,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, 2013, pp. 235–245.

[152] J. Dörre, S. Apel, and C. Lengauer, “Static type checking of Hadoop MapReduce programs,” *Proc. Second Int. Work. MapReduce its Appl. - MapReduce ’11*, p. 17, 2011.

[153] A. Rabkin and R. Katz, “Static extraction of program configuration options,” *2011 33rd*

Int. Conf. Softw. Eng., pp. 131–140, 2011.

[154] K. Ono, Y. Hirai, Y. Tanabe, N. Noda, and M. Hagiya, “Using Coq in specification and program extraction of Hadoop MapReduce applications,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2011, vol. 7041 LNCS, pp. 350–365.

[155] R. Hamlet, *Random testing*, no. 1. Wiley, 1994.

[156] J. M. Glenford, *The art of software testing*. 1979.

[157] M. Grindal, J. Offutt, and S. F. Andler, “Combination testing strategies: A survey,” *Softw. Test. Verif. Reliab.*, vol. 15, no. 3, pp. 167–199, 2005.

[158] C. Nie and H. Leung, “A survey of combinatorial testing,” *ACM Comput. Surv.*, vol. 43, no. 2, pp. 1–29, 2011.

[159] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, “The case for evaluating mapreduce performance using workload suites,” in *IEEE International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems - Proceedings*, 2011, pp. 390–399.

[160] “GridMix: benchmark for Hadoop clusters.” [Online]. Available: https://hadoop.apache.org/docs/stable1/gridmix.html.

[161] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, “SparkBench: a spark benchmarking suite characterizing large-scale in-memory data analytics,” *Cluster Comput.*, vol. 20, no. 3, pp. 2575–2589, 2017.

[162] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolette, and H.-A. Jacobsen, “Bigbench: Towards an industry standard benchmark for big data analytics,” *Proc. 2013 ACM SIGMOD Int. Conf. Manag. Data*, pp. 1197–1208, 2013.

[163] “TPCx-BB: TPC Big Data Benchmark.” [Online]. Available: http://www.tpc.org/tpcx-bb/.

[164] C. Wohlin, P. Runeson, P. A. Da Mota Silveira Neto, E. Engström, I. Do Carmo Machado, and E. S. De Almeida, “On the reliability of mapping studies in software engineering,” *J. Syst. Softw.*, vol. 86, no. 10, pp. 2594–2610, 2013.

[165] L. Major, T. Kyriacou, and O. P. Brereton, “Systematic literature review: Teaching novices programming using robots,” *IET Softw.*, vol. 6, no. 6, pp. 502–513, 2012.

[166] R. L. Bocchino, V. S. Adve, S. V Adve, M. Snir, and R. L. B. Jr., “Parallel Programming Must Be Deterministic by Default,” *Proc. First USENIX Conf. Hot Top. parallelism*, vol. 22, no. 1, p. 4, 2009.

[167] “Average temperature per year.” [Online]. Available: https://github.com/t2013anurag/Hadoop-Map-Reduce-Avg-Temp.

[168] “Average temperature per year.” [Online]. Available: https://github.com/hanasu/ClimateData.

[169] J. Lin and C. Dyer, “Data-Intensive Text Processing with MapReduce,” *Synth. Lect. Hum. Lang. Technol.*, vol. 3, no. 1, pp. 1–177, 2010.

[170] “Open Ankus: Data mining and machine learning based on MapReduce.” [Online]. Available: http://www.openankus.org/.

[171] A. Orso and G. Rothermel, “Software testing: a research travelogue (2000–2014),” *Proc.*

Futur. Softw. Eng. - FOSE 2014, pp. 117–132, 2014.

[172] 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.

[173] D. Hamlet and R. Taylor, “Partition Testing Does Not Inspire Confidence,” *IEEE Trans. Softw. Eng.*, vol. 16, no. 12, pp. 1402–1411, 1990.

[174] J. W. Duran and S. C. Ntafos, “An Evaluation of Random Testing,” *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 438–444, 1984.

[175] A. Arcuri, M. Z. Iqbal, and L. Briand, “Random testing: Theoretical results and practical implications,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 258–277, 2012.

[176] W. J. Gutjahr, “Partition testing vs. random testing: The influence of uncertainty,” *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 661–674, 1999.

[177] A. W. Williams and R. L. Probert, “A measure for component interaction test coverage,” in *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA*, 2001, vol. 2001–Janua, pp. 304–311.

[178] P. Ammann and J. Offutt, “Using formal methods to derive test frames in category-partition testing,” *Comput. Assur. 1994. COMPASS ’94 Safety, Reliab. Fault Toler. Concurr. Real Time, Secur. Proc. Ninth Annu. Conf.*, pp. 69–79, 1994.

[179] D. R. Kuhn and M. J. Reilly, “An investigation of the applicability of design of experiments to software testing,” in *Proceedings - 27th Annual NASA Goddard / IEEE Software Engineering Workshop, SEW 2002*, 2003, pp. 91–95.

[180] J. Huller, “Reducing time to market with combinatorial design method testing,” in *IN PROCEEDINGS OF THE 2000 INTERNATIONAL COUNCIL ON SYSTEMS ENGINEERING (INCOSE) CONFERENCE*, 2000, pp. 16--20.

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

[182] 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.

[183] 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.

[184] 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.

[185] 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.

[186] 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.

[187] R. W. Selby and V. R. Basili, “Comparing the Effectiveness of Software Testing Strategies,” *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 12, pp. 1278–1296, 1987.

[188] H. A. De Souza, M. L. Chaim, and F. Kon, “Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges,” *arxiv16*, pp. 1–40, 2016.

[189] S. B. Kotsiantis, “Supervised Machine Learning: A Review of Classification Techniques,” *Informatica*, vol. 31, pp. 249–268, 2007.

[190] V. R. Basili and H. Dieter Rombach, “The TAME Project: Towards Improvement-Oriented Software Environments,” *IEEE Trans. Softw. Eng.*, vol. 14, no. 6, pp. 758–773, 1988.

[191] B. Rivas, J. Merino, M. Serrano, I. Caballero, and M. Piattini, “I8K|DQ-BigData: I8K architecture extension for data quality in big data,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015, vol. 9382, pp. 164–172.

[192] “Movies analysis implemented in MapReduce.” [Online]. Available: https://github.com/adityaundirwadkar/mapreduce-programming/tree/master/example_1.

[193] “Treelogic S.L.” [Online]. Available: www.treelogic.com.

[194] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*, vol. 9783642290. 2012.

[195] “MapReduce algorithm of Connected components in graphs.” [Online]. Available: https://github.com/Draxent/ConnectedComponents.

[196] “Goldstein analysis implemented in MapReduce.” [Online]. Available: https://github.com/tchira/MapReduce/tree/master/src/main/java/com/telenav/hadoop/gratio.

[197] “Analysis of the New York restaurants based on MapReduce.” [Online]. Available: https://github.com/Shubham617/MapReduce-Project/tree/master/NYC Restaurant Data.

[198] T. D. Cook and D. T. (Donald T. Campbell, *Quasi-experimentation : design &amp; analysis issues for field settings*. Houghton Mifflin, 1979.

[199] R. Malhotra, *Empirical research in software engineering : concepts, analysis, and applications*. .

[200] A. S. Namin and S. Kakarla, “The use of mutation in testing experiments and its sensitivity to external threats,” *Proc. 2011 Int. Symp. Softw. Test. Anal. - ISSTA ’11*, p. 342, 2011.

[201] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, “Threats to the validity of mutation-based test assessment,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, 2016, pp. 354–365.

[202] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?,” in *Proceedings of the 27th international conference on Software engineering - ICSE ’05*, 2005, p. 402.

[203] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, 2014, pp. 654–665.

[204] R. Guderlei and J. Mayer, “Statistical Metamorphic Testing - Testing programs with random output by means of statistical hypothesis tests and Metamorphic Testing,” in *Proceedings - International Conference on Quality Software*, 2007, pp. 404–409.

[205] C. Murphy and G. Kaiser, “Empirical evaluation of approaches to testing applications

without test oracles," *Dep. Comput. Sci. Columbia Univ. Tech. Rep. CUCS-039-09*, 2010.

[206] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*, 2011, p. 199.

[207] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Trans. Softw. Eng.*, vol. PP, no. 99, pp. 1–1, 2016.

[208] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," *ACM SIGSOFT Softw. Eng. Notes*, vol. 22, no. 6, pp. 432–449, 1997.

[209] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "Empirical investigation of the relationship between spectra differences and regression faults," *Softw. Test. Verif. Reliab.*, vol. 10, no. 3, pp. 171–194, 2000.

[210] R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings - Testing: Academic and Industrial Conference Practice and Research Techniques, TAIC PART-Mutation 2007*, 2007, pp. 89–98.

[211] L. Naish, H. J. Lee, and K. Ramamohanarao, "Spectral debugging with weights and incremental ranking," in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2009, pp. 168–175.

[212] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Inform.*, vol. 18, no. 1, pp. 31–45, 1982.

[213] W. Masri and R. A. Assi, "Prevalence of coincidental correctness and mitigation of its impact on fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 1–28, 2014.

[214] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist," *RN Univ. Coll. London*, 2014.

[215] A. Arrieta, S. Segura, U. Markiegi, G. Sagardui, and L. Etxeberria, "Spectrum-based fault localization in software product lines," *Inf. Softw. Technol.*, 2018.

[216] D. Lewis, "Causation," *J. Philos.*, vol. 70, no. 17, p. 556, 1973.

[217] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error explanation with distance metrics," in *International Journal on Software Tools for Technology Transfer*, 2006, vol. 8, no. 3, pp. 229–247.

[218] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 1–32, 2011.

[219] M. Harman and B. F. Jones, "Search-based software engineering," *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, 2001.

[220] D. E. Goldberg, "Optimal initial population size for binary-coded genetic algorithms," *Clear. Genet. Algorithms, Dep. Eng. Mech. Univ. Alabama*, 1985.

[221] J. D. Schaffer, R. A. Caruana, L. J. Eshelman, and R. Das, "A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization," *Proc. 3rd Int. Conf. Genet. Algorithms*, vol. 3, pp. 51–60, 1989.

[222] T. Bäck, F. Hoffmeister, and H.-P. Schwefel, "A survey of evolution strategies," *Proc.*

Fourth Int. Conf. Genet. Algorithms, vol. 9, no. 3, p. 8, 1991.

[223] R. Le Riche, C. Knopf-Lenoir, and R. Haftka, “A Segregated Genetic Algorithm for Constrained Structural Optimization.,” in *6th International Conference on Genetic Algorithms*, 1995, pp. 558–565.

[224] K. A. De Jong, “Analysis of the behavior of a class of genetic adaptive systems,” University of Michigan, 1975.

[225] J. Cantó, S. Curiel, and E. Martínez-Gómez, “A simple algorithm for optimization and model fitting: AGA (asexual genetic algorithm),” *Astron. Astrophys.*, vol. 501, no. 3, pp. 1259–1268, 2009.

[226] M. L. Mauldin, “Maintaining diversity in genetic search,” in *National Conference on Artificial Intelligence*, 1984, pp. 247–250.

[227] J. Grefenstette, “Optimization of Control Parameters for Genetic Algorithms,” *IEEE Trans. Syst. Man. Cybern.*, vol. 16, no. 1, pp. 122–128, 1986.

[228] Y. W. Leung, Y. Wang, Y. W. Leung, and Y. Wang, “An orthogonal genetic algorithm with quantization for global numerical optimization,” *IEEE Trans. Evol. Comput.*, vol. 5, no. 1, pp. 41–53, 2001.

[229] W. E. Wong, V. Debroy, and D. Xu, “Towards better fault localization: A crosstab-based statistical approach,” *IEEE Trans. Syst. Man Cybern. Part C Appl. Rev.*, vol. 42, no. 3, pp. 378–396, 2012.

[230] W. E. Wong and Y. Qi, “Bp Neural Network-Based Effective Fault Localization,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 19, no. 4, pp. 573–597, 2009.

[231] B. Jiang, Z. Zhang, W. K. Chan, T. H. Tse, and T. Y. Chen, “How well does test case prioritization integrate with statistical fault localization?,” *Inf. Softw. Technol.*, vol. 54, no. 7, pp. 739–758, 2012.

[232] S. McMaster and A. Memon, “Call-stack coverage for GUI test suite reduction,” *IEEE Trans. Softw. Eng.*, vol. 34, no. 1, pp. 99–115, 2008.

[233] S. McMaster and A. M. Memon, “Call stack coverage for test suite reduction,” in *IEEE International Conference on Software Maintenance, ICSM*, 2005, vol. 2005, pp. 539–548.

[234] S. Sprenkle, S. Sampath, E. Gibson, L. Pollock, and A. Souter, “An Empirical Comparison of Test Suite Reduction Techniques for User-Session-Based Testing of Web Applications,” in *ICSM ’05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, 2005, pp. 587–596.

[235] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, “Effect of test set minimization on fault detection effectiveness,” in *Proceedings of the 17th international conference on Software engineering - ICSE ’95*, 1995, pp. 41–50.

[236] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, “An empirical study of the effects of minimization on the fault\ndetection capabilities of test suites,” *Proceedings. Int. Conf. Softw. Maint. (Cat. No. 98CB36272)*, 1998.

[237] D. Fisher, R. DeLine, M. Czerwinski, and S. Drucker, “Interactions with big data analytics,” *interactions*, vol. 19, no. 3, p. 50, 2012.

[238] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, “Mochi: Visual Log-Analysis Based Tools for Debugging Hadoop,” in *Proc. of the HotCloud - Conf. on Hot Topics in*

Cloud Computing, 2009, pp. 1–5.

[239] E. Garduno, S. P. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, “Theia: visual signatures for problem diagnosis in large hadoop clusters,” in *Proceedings of the International Conference on Large Installation System Administration*, 2012, no. 2, pp. 33–42.

[240] N. Khoussainova, M. Balazinska, and D. Suciu, “PerfXplain: debugging MapReduce job performance,” *Proc. VLDB Endow.*, vol. 5, no. 7, pp. 598–609, Mar. 2012.

[241] C. Olston and B. Reed, “Inspector Gadget: A framework for custom monitoring and debugging of distributed dataflows,” *Proc. 2011 ACM SIGMOD Int. Conf. Manag. data*, pp. 1221–1224, 2011.

[242] V. Jagannath, Z. Yin, and M. Budiu, “Monitoring and debugging DryadLINQ applications with daphne,” in *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011, pp. 1266–1273.

[243] M. A. Gulzar, M. Interlandi, X. Han, M. Li, T. Condie, and M. Kim, “Automated debugging in data-intensive scalable computing,” in *Proceedings of the 2017 Symposium on Cloud Computing - SoCC ’17*, 2017, pp. 520–534.

[244] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, “Test-case reduction for C compiler bugs,” in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation - PLDI ’12*, 2012, p. 335.

[245] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein, and M. Kim, “BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark,” *Proc. 38th Int. Conf. Softw. Eng. - ICSE ’16*, vol. 2016, pp. 784–795, 2016.

[246] A. Dave, M. Zaharia, S. Shenker, and I. Stoica, “Arthur: Rich Post-Facto Debugging for Production Analytics Applications,” 2013.

[247] D. Logothetis, S. De, and K. Yocum, “Scalable lineage capture for debugging DISC analytics,” in *Proceedings of the 4th annual Symposium on Cloud Computing - SOCC ’13*, 2013, pp. 1–15.

[248] E. Bergen and S. Edlich, “Post-Debugging in Large Scale Big Data Analytic Systems,” in *Datenbanksysteme für Business, Technologie und Web*, 2017, pp. 65–74.

[249] 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.

[250] 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.

[251] 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.

[252] 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.

[253] 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.

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

[255] 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.

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

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

[258] 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.

[259] “Mockito: Tasty mocking framework for unit tests in Java.” [Online]. Available: http://mockito.org/. [Accessed: 24-Feb-2017].

[260] “PowerMock: Mocking framework for unit testing.” [Online]. Available: http://powermock.github.io/. [Accessed: 24-Feb-2017].

[261] “HBase: Hadoop database, a distributed, scalable, big data store.” [Online]. Available: https://hbase.apache.org/.

[262] 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.

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

[264] “SequenceFile: a flat file consisting of binary key/value pairs.” [Online]. Available: https://wiki.apache.org/hadoop/SequenceFile. [Accessed: 24-Feb-2017].

[265] 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.

[266] 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.

[267] 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.

[268] “Hamcrest: Matchers that can be combined to create flexible expressions of intent.” [Online]. Available: http://hamcrest.org/. [Accessed: 02-Mar-2017].

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

[270] 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.

[271] 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.

[272] 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.

[273] “Jetty server.” [Online]. Available: http://www.eclipse.org/jetty/. [Accessed: 23-Feb-2017].

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

[275] “SQLite: a self-contained, high-reliability, embedded, full-featured, public-domain, SQL database engine.” [Online]. Available: https://sqlite.org. [Accessed: 23-Feb-2017].

[276] “Google Scholar.” [Online]. Available: https://scholar.google.es/. [Accessed: 27-Feb-2018].