TRANSMUT-SPARK: Transformation Mutation for Apache Spark

João Batista de Souza Neto, Anamaria Martins Moreira, Genoveva Vargas-Solar, Martin A Musicante

TRANSMUT-SPARK: Transformation Mutation for Apache Spark - Long Version

Genoveva Vargas-Solar, João Batista de Souza Neto, Anamaria Martins Moreira, Martin Musicante, João Batista, Souza Neto

TRANSMUT-SPARK: Transformation Mutation for Apache Spark

João Batista de Souza Neto[1,4*], Anamaria Martins Moreira[2], Genoveva Vargas-Solar[3] and Martin A. Musicante[1]

[1]*Department of Informatics and Applied Mathematics (DIMAp), Federal University of Rio Grande do Norte (UFRN), Natal, Brazil.*

[2]*Computer Science Department (DCC), Federal University of Rio de Janeiro (UFRJ), Rio de Janeiro, Brazil.*

[3] *French Council of Scientific Research (CNRS), LIRIS, Lyon, France.*

[4] *Department of Informatics, Management and Design (DIGD-DV), Federal Center for Technological Education of Minas Gerais, Divinópolis, Brazil.*

SUMMARY

This paper proposes TRANSMUT-SPARK for automating mutation testing of Big Data processing code within Spark programs. Apache Spark is an engine for Big Data Analytics/Processing that hides the inherent complexity of parallel Big Data programming. Nonetheless, programmers must cleverly combine Spark built-in functions within programs and guide the engine to use the right data management strategies to exploit the computational resources required by Big Data processing and avoid substantial production losses. Many programming details in Spark data processing code are prone to false statements that must be correctly and automatically tested. This paper explores the application of mutation testing in Spark programs, a fault-based testing technique that relies on fault simulation to evaluate and design test sets. The paper introduces TRANSMUT-SPARK for testing Spark programs by automating the most laborious steps of the process and fully executing the mutation testing process. The paper describes how the TRANSMUT-SPARK automates the mutants generation, test execution, and adequacy analysis phases of mutation testing. It also discusses the results of experiments to validate the tool and argues its scope and limitations.

Received . . .

DOI: 10.1002/stvr

Softw. Test. Verif. Reliab. 2010; **00**:2–61
KEY WORDS: Apache Spark; Mutation Testing; Testing Tool; TRANSMUT-SPARK
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

1. INTRODUCTION

The Big Data phenomenon has introduced a series of processing challenges using greedy algorithms. These algorithms call for well-adapted processing infrastructures and programming models to support their execution on large-scale clustered architectures. Existing frameworks adopt either control flow [1, 2] or data flow approaches [3, 4, 5]. Apache Spark [5] has emerged as one of the main data-flow engines for parallel Big Data processing. It screens difficulties inherent to parallel programming and distributed processing, allowing programmers to focus only on the algorithmic aspects of parallel programs. Spark's data flow programming model represents programs as directed acyclic graphs (DAGs) that coordinate the execution of operations applied on datasets distributed across several computing nodes.

Even with more minor painful ways of designing, programming, and executing big data parallel processing programs, programmers still need to tune several aspects within their code and configure resource allocation. The complexity of this tuning task is due to the number of aspects to consider and can lead to errors. Therefore, testing programs becomes essential to avoid losses in production [6]. In this context, software testing techniques emerge as relevant tools [7, 8]. This paper addresses the testing of big data processing code weaved within Spark programs by exploring the application of *mutation testing* [9].

Mutation testing is a powerful test technique where tests are designed to pinpoint specific faults introduced in the code, the *mutations*. The quality of the resulting tests is closely dependent on how representative these faults are for the programs developed in that specific language or programming paradigm. In a previous paper [10] we proposed a *transformation mutation* approach that applies mutation testing in Spark programs introducing mutation operators designed to simulate faults

*Correspondence to: João Batista de Souza Neto, Department of Informatics and Applied Mathematics (DIMAp), Federal University of Rio Grande do Norte (UFRN), Natal, Brazil. E-mail: jbsneto@ppgsc.ufrn.br

specific to data flow programs. We conducted manual experiments to show our mutation operators applicability to test Spark programs. Even if results were promising, we confirmed that mutation testing could be laborious, time-consuming and prone to errors [11] if it is not partially automated. We believe that mutation testing must be delegated to a tool to perform intensive testing and face software development conditions that imply production speed and quality requirements. Therefore, we propose TRANSMUT-SPARK that automates the Spark programs' mutation testing process.

TRANSMUT-SPARK implements requirements that a mutation test tools must provide [12]: *test case handling*, including the execution, inclusion, and exclusion of test cases; *mutant handling*, for generating, selecting, executing and analyzing mutants; and *adequacy analysis* for calculating the mutation score and generating reports. The paper shows how TRANSMUT-SPARK automates the primary and most laborious steps of the process and enables the full execution of the mutation testing process for Spark programs. These features were thoroughly validated through a new complete round of experiments discussed in the paper and that enhance those described in a previous paper [10]. Additionally, TRANSMUT-SPARK was experimentally compared with traditional mutation testing done at the (syntactic) programming language level. Experiments showed that TRANSMUT-SPARK is complementary to mutation testing for Scala programs that do not address Big Data processing code. Thus, combining mutation testing approaches addressing programming languages statements with TRANSMUT-SPARK can test both higher-level Big Data processing code that is weaved within classic imperative programs and the more basic program constructs. The contribution presented in this paper is twofold: (1) TRANSMUT-SPARK for automating mutation testing of Big Data processing code weaved within Scala programs, (2) and an experimental battery of tests evaluating the possibilities of mutation testing for Spark programs that could not be carried out without the tool.

Accordingly, the remainder of the paper is organized as follows. Section 2 introduces the background concepts behind TRANSMUT-SPARK namely Apache Spark and mutation testing. Section 3 introduces related work concerning approaches addressing the problem of testing programs implemented to run on top of big data processing platforms. Section 4 introduces the set

Prepared using stvrauth.cls
DOI: 10.1002/stvr

of mutation operators we propose for Apache Spark programs. Section 5 introduces TRANSMUT-SPARK the tool that we propose for automating the process of testing Spark programs written in Scala. Section 6 describes the experiments that we conducted for evaluating TRANSMUT-SPARK as a testing tool. It also describes a comparative study on transformation mutation and traditional mutation testing. Section 7 summarizes, analyses and discusses experimental results. Section 8 discusses the limitations and threats to validity. Finally, Section 9 concludes the paper and discusses future work. https://www.overleaf.com/project/618a6bf5ad33492c9e33c0bc

2. BACKGROUND

The approach behind TRANSMUT-SPARK relies on two conceptual underpinnings: Apache Spark and mutation testing. This section introduces the fundamental concepts of these components underlying those that are key to understand the transformation mutation strategy for Big Data processing code in Spark promoted by TRANSMUT-SPARK.

2.1. Apache Spark

Apache Spark [5] is an execution platform for large-scale data processing parallel programs written in programming languages like Scala, Java, Python, and R. It is suited for embedding machine learning algorithms and interactive analysis, such as exploratory queries on datasets within programs. It offers libraries for working with structured data using an SQL-style API (*SparkSQL*), machine learning (*MLlib*), streaming data processing (*Spark Streaming*) and graph processing (*GraphX*) [13].

Spark prevents programmers from dealing with data distribution among cluster nodes and ensuring fault tolerance and processes' synchronization. For dealing with fault-tolerant data distribution, Spark is based on a data abstraction called *Resilient Distributed Dataset* (RDD) [14]. An RDD represents a collection of data distributed through the nodes of a cluster that can be processed in parallel within a Spark program. Fault-tolerance for RDD's means that its partitions can be reconstructed in case of failures emerging at execution time.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

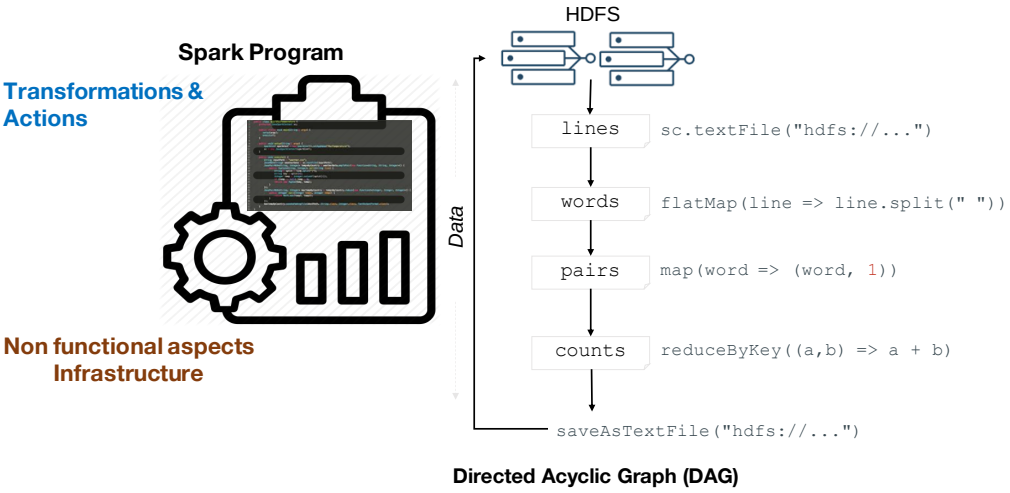


Figure 1. Spark program overview.

The general structure of a Spark program is shown in Figure 1. It consists of two types of blocks: (1) functional aspects expressed by a data flow implementing the application logic; (2) operations devoted to processing data (transformations) and guiding the way data is managed from memory to cache and disk (actions).

As shown in Figure 1, the data used by a program are initially stored in a persistence support, often in a distributed file system, for example, on HDFS (Hadoop Distributed File System) [1]. The data are first retrieved from the persistence support and transformed into an RDD for executing a Spark program. The data processing operations stated in a Spark program will then be applied to this RDD. For example, the program in the figure is intended to count the occurrence of words in a text dataset. The program runs through the dataset to first separate the lines of text into separate words. So, for each word, a key/value tuple is created containing the word as key and the integer 1 as value. Afterwards, key/value tuples are grouped by the key, and the values are aggregated, resulting in a dataset with the words and their frequencies, *i.e.*, the number of times a word appears in the text. Finally, the resulting dataset is stored (e.g., HDFS). The sequence of operations applied in a Spark program forms a DAG (Directed Acyclic Graph) representing the program's execution plan optimized by Spark when executed in a cluster.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

In the program there are two classes of operations: *transformations* and *actions*. Understanding the way the Spark platform executes them is essential to understand the implications of using them within a program.

Transformations: create new RDDs from existing ones [13]. In Spark, transformations are evaluated under a *lazy evaluation* [5] strategy [†], that is until other computations need them. Spark transformations can receive functions as input parameters and apply them over the RDDs elements to generate a new RDD.

Spark's transformations can be classified into the following types: (i) mapping (such as *map* and *flatMap*), apply a function to map the values of an input RDD to an output RDD; (ii) filtering, filter the values of an RDD based on a predicate function; (ii) grouping (*groupBy*, *groupByKey*), group the values of an RDD based on a key; (iii) aggregation (*reduceByKey*, *aggregateByKey*), aggregate values grouped by a key; (iv) set (*union*, *intersection*, *subtract*, *distinct*), similar to mathematical set operations; (v) join (*join*, *leftOuterJoin*, *rightOuterJoin*, *fullOuterJoin*), join two RDDs on the basis of a common key; (vi) sort (*sortBy*, *sortByKey*), sort the values in an RDD.

Transformations can be either *narrow* or *wide*, depending on the degree of distribution of the data sets [14]. When each partition at the parent RDD is used by most of the child RDD partition, there is a narrow dependency. Operations like *map*, *flatMap* and *filter* are examples of narrow transformations. In wide transformations, all the required elements to compute the records in the single partition may live in many partitions of parent RDD. Generally, this type of transformation operates on key/value tuple datasets that need data grouped by key, such as *reduceByKey* and *join* operations. This type of operation causes a reorganization of the data in the cluster to group values that have the same key to the same partition. This process of data reorganization is called *shuffling*. This mechanism involves copying and sending data between the cluster nodes, making it a complex and costly process.

[†] In programming language theory, lazy evaluation, or call-by-need, is an evaluation strategy that delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations (sharing).

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
1  val input = sc.textFile("hdfs://...")
2  val words = input.flatMap( (line: String) => line.split(" ") )
3  val pairs = words.map( (word: String) => (word, 1) )
4  val counts = pairs.reduceByKey( (a: Int, b: Int) => a + b )
5  counts.saveAsTextFile("hdfs://...")
```

Figure 2. Example of a Spark program.

Actions: return values that are *not* RDDs to the Driver or write the content of the RDD in some storage system. Actions trigger the lazy evaluation execution of transformations. In this way, Spark can optimize the execution of applications by executing narrow transformations in the same process (pipeline) and create different stages for wide operations that trigger the shuffling process [14].

A Spark program execution is coordinated by a *Driver* through a *SparkContext*. The *SparkContext* executes the main program function consisting of a sequence of operations on RDDs and manages internal program information and deploys the operations on the cluster nodes [15].

Some of the main Spark actions are *reduce*, which aggregates all the values of an RDD into a single value; *collect*, which returns to the Driver the contents of an RDD in the form of an array; and *saveAsTextFile*, which saves the content of an RDD in a storage system.

Spark program example Figure 2 shows the code of the counting words example programmed in Apache Spark using the Scala programming language. The program starts by reading a data set and storing it in the *input* variable (line 1). Next, it applies the *flatMap* transformation, which separates each line of text into words (line 2). The RDD with words is transformed into a key/value RDD with the application of the transformation *map* which generates key/value pairs in which the key is a word, and the value is the integer number 1 (line 3). The counting of words is done applying the *reduceByKey* transformation, which groups the values per key. Then, it applies the (associative) addition function to sums all the values, resulting in the RDD *count*, containing the frequency of each word (line 4). The program terminates by calling the action *saveTextFile* which performs the transformations and saves its result (line 5).

A particular characteristic of a Spark program is that it includes operations devoted to processing data (application logic) and operations devoted to guiding the way data are managed to execute the program (*i.e.*, data swapped from disk to cache and memory, shared or replicated across different

Prepared using stvrauth.cls
DOI: 10.1002/stvr

processes and nodes, transmitted across processes). Data management decisions are independent of the application logic but have a significant impact on its performance. Designing a Spark program implies ensuring functional faults are inconsistencies within the code that may result in unexpected behaviour when executed, for example, wrong transformations, incorrect parameters, and absence of transformations. In this paper we focus on these types of faults (see Section 4).

2.2. *Mutation Testing*

De Millo *et al.* [9] proposed a fault-based testing technique named *mutation testing*. It consists of creating variants of a program and then deriving tests that show that the variants behave differently from the original program. These variants, called *mutants*, are obtained from the program to be tested by performing minor modifications that simulate common faults or programmers mistakes. These modifications are systematically created by using predefined rules, called *mutation operators*.

In mutation testing tests must be designed to identify mutants from an original program. A test should identify that the result obtained with a mutant is different from the result obtained with the original program. When this occurs, the test is said to *kill* the mutant. A mutant and the original program can have the same behavior; thus, the mutant cannot be killed. The mutant is said to be *equivalent*, and it is removed from the test requirements set. The test developer can then use each non-equivalent mutant as a guide to derive interesting test cases (*i.e.*, input test data that will exercise the program in such a way that if each of those specific faults were to be in the code, there would be a test case in the test set that is capable of showing its presence to the tester). A typical example is a mutant where a conditional construct guarded by a comparison of the kind $a < b$ is changed into $a \leq b$. Unless we have a test case where a and b have the same value at this point, the mistake will remain unnoticed.

Mutation testing coverage as a testing requirement can be determined from the ratio of the number of killed mutants to the total number of mutants, not considering equivalent mutants. This ratio is known as *mutation score*, being used as a quality measure for test sets [16]. The following formula

Prepared using stvrauth.cls
DOI: 10.1002/stvr

calculates the mutation score:

$$ms(P,T) = \frac{DM(P,T)}{M(P) - EM(P)}$$

where $ms(P,T)$ is the mutation score for a program P and a test set T, $DM(P,T)$ is the number of mutants of the program P killed by the test set T, $M(P)$ is the number of mutants generated from P, and $EM(P)$ is the number of mutants equivalent to P.

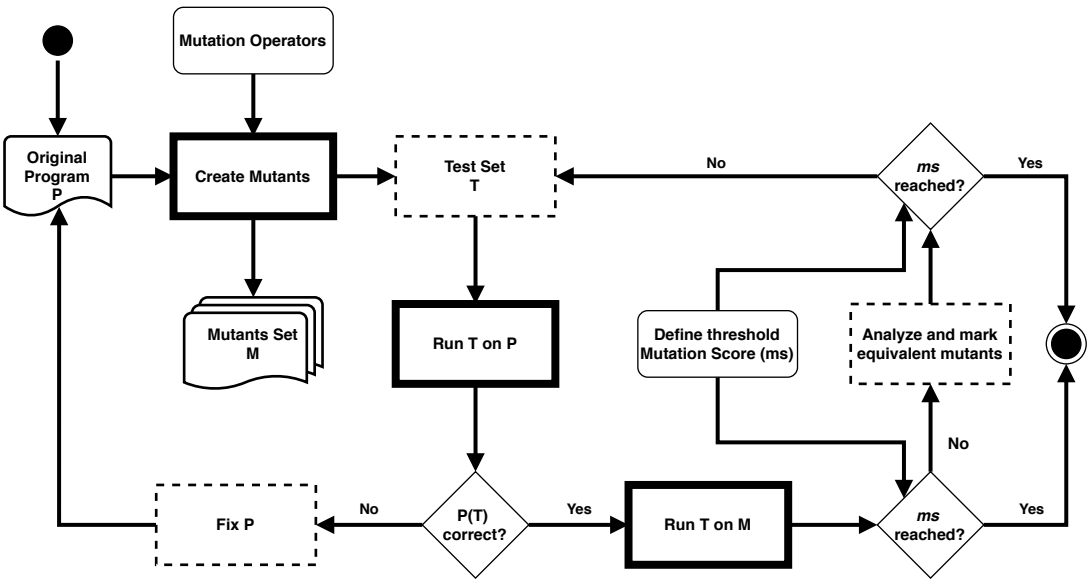


Figure 3. Mutation testing process (adapted from a proposal by Ammann and Offut [16]).

Applying mutation testing to a program involves the following steps: *generation of mutants*, *execution of the original program*, *execution of mutants*, and *analysis of living mutants* to determine whether or not they are equivalent [16]. This process is shown in Figure 3, the automated steps are represented in bold boxes, and the manual steps are in dashed boxes. Given a program P, a set of mutants M is created from the application of mutation operators in P. Then, a set of test cases T is created, which can be designed based on the mutants in M or from another testing criteria based on the choice of the test engineer. This test set T is then run with the program P. If the tests fail, it is necessary to fix program P. Otherwise, T is run with the mutants in M. The mutation score (ms) is calculated based on the number of dead and alive mutants with the test results. From that step, it is necessary to decide whether or not the mutation testing process should continue. This decision

Prepared using stvrauth.cls
DOI: 10.1002/stvr

can be made based on the value of ms; if it is 1.0, it means that all non-equivalent mutants have been killed by the test set T. However, it is not always feasible to kill all mutants, given that many mutants can be created. Thus, at the discretion of the test engineer, it is possible to define a threshold value for ms, generally close to 1.0, so that when this value is reached, the test set T is considered good, so the process ends.

Mutation testing is one of the most challenging criteria to meet, given the significant effort required to apply all of its steps, as shown in Figure 3. Given this process and the large number of mutants generated, it is essential that mutation testing be supported by a tool that automates all or part of the process. It is impractical to apply it manually in production.

Over the years, mutation testing has been actively investigated in the area of software testing research [17]. Studies have identified mutation testing as one of the most efficient criteria for detecting faults compared to other criteria, such as the works of Franklin et al. [18], Offut et al. [19] and Walsh [20]. These results make mutation testing a reference in the software testing area and are often used as a quality standard to evaluate other criteria, techniques, and test sets in general [16]. As usual, with very flexible and powerful techniques, mutation testing success depends on how this flexibility is instantiated to each situation. A good set of mutation operators is needed to design tests that explore programs' main potential problems in that specific language or paradigm. Mutation testing for imperative programs requires different mutation operators than those used for a functional language; similarly, an object-oriented program requires some mutation operators to deal with particularities of object-oriented programming. Besides, in frameworks like Apache Spark, Big Data processing code is weaved within a host programming language (e.g., Scala) code. This approach introduces a specific challenge for testing approaches like mutation, which must test "classic" code in the host language and propose specific mutation strategies to test the Big Data processing code. Our work addresses mutation testing for the parallel Big Data processing code weaved within Scala programs in the context of Apache Spark like frameworks.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

3. RELATED WORK

Parallel programming models are based on the principle that significant problems can often be divided into smaller ones, which can then be solved simultaneously. Regarding big data processing, two strategies have emerged, namely *control flow* and *data flow*-based parallel programming. Under the control flow strategy, a single system node controls the entire program execution (master, orchestrator). In the data flow parallel model, the processes that execute the program trigger the execution of other program components. Several databases projects focus on data flow control models since they have good response times and throughput [21].

The most popular control flow-based model is MapReduce [22]. A MapReduce program has two main components: A *map* procedure, which applies the same function to all the elements of a key/value dataset, and a *reduce* procedure, which aggregates values based on their key. The map component filters or sorts data (such as sorting students by the first name into queues, one queue for each name). The reduce component usually summarizes data (like counting the number of students in each queue, yielding name frequencies). MapReduce implementations such as Hadoop [1] enhance reliability by supporting the construction of fault-tolerant systems.

Big Data processing programs reliability is essential because failures can generate large losses [6] given the large number of computational resources required for their execution. In this context, software testing becomes paramount in developing programs with higher quality and less failure-prone during production. The studies presented by Camargo *et al.* [23] and Morán *et al.* [24] show that works on testing in this context have increased interest in recent years. They reveal that few systematic techniques and tools have been developed so far and that most have not reached a certain level of maturity. Furthermore, most of the existing work has focused on programs that exclusively follow control flow based parallel programming models like *MapReduce* [24], showing that the testing of Big Data processing programs is still an open research area. This section presents the state of the art of functional testing of large data processing programs.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

The systematic review presented by Morán *et al.* [24] identified 54 works related to the validation and verification process of control flow-based programming models, mainly based on the MapReduce model. The study revealed that most areas focus on program performance, identifying 32 (59%) works related to performance testing. In contrast, only 12 (22%) works are focused on functional testing, indicating a potentially promising subject in the area.

Works in the performance testing domain involve verifying non-functional requirements, such as execution time and computational resources. Functional testing is related to the program's functional behaviour. We focus on functional testing of Big Data processing programs. We do not analyze works on performance testing since they are mostly related to benchmarks, which is out of the scope of our work.

We describe the works that test parallel data processing programs verifying their functional behaviour to check whether their functional requirements are met.

3.1. Testing MapReduce (control flow-based) Programs

Testing MapReduce programs can be done dynamically (*i.e.*, observing programs behaviour at runtime) and statically (*i.e.*, analyzing the source code). The following sections describe some dynamic and static testing approaches for MapReduce programs implemented on top of different platforms of the Apache Hadoop ecosystem[‡].

Dynamic testing of control flow-based programs. The general principle is to execute a program to verify its properties. The work presented by Csallner *et al.* [25] uses a symbolic execution technique to search for faults and generate test cases. Therefore, they extract algebraic expressions from the program representing the conditions that lead to different execution paths. From this, they derive coded execution paths with correction conditions (which verify commutativity). Then, they use a constraint solver to infer input values that violate the correction conditions. Input values are then converted into test cases for the program. The testing technique has been implemented for

[‡]The Apache Hadoop project develops open-source software for reliable, scalable, distributed computing. The Hadoop software library is a framework that allows for the distributed processing of large datasets across clusters of computers using simple programming models (`https://hadoop.apache.org`).

Prepared using stvrauth.cls
DOI: 10.1002/stvr

MapReduce programs in Hadoop. A similar approach was applied by Li *et al* [26] to test programs in *Pig Latin* [27], a *script* language for data processing programs that run on Hadoop. The authors expand the technique developed by Csallner *et al.* [25] to extract test paths by exploring the different operations of Pig Latin.

Xu *et al.* [28] introduce a technique that tests properties in operations of stream processing programs with the *stream processing* language *SPL* [29]. The technique seeks to verify properties essential for the reliability and optimization of this type of program: non-determinism, selectivity, blocking, statefulness, non-commutativity and partition interference. The technique uses random data generation to dynamically check (with execution) whether or not these properties are verified.

MRTree [30] is a hierarchical classification of faults for MapReduce programs. The faults presented in the classification are divided into faults related to the *reduce* operation and the *combine* operation (an intermediate reduction operation performed right after *map*). Some examples of the faults presented in *MRTree* are: (1) a non-commutative reduce operation that can generate different results for the same dataset if data is processed in different orders; and (2) faults related to the key/value data, such as inconsistencies between the key and the value or the issue of an incorrect key/value pair, are checked. For each fault, the authors propose *ad-hoc* testing directives to mitigate them. The same research group defined *MRFlow* [31], a technique that derives a data flow graph from *map* and *reduce* operations. From this graph, test cases are generated using *graph-based testing techniques*. In both papers [30, 31] authors do not describe a tool implementing their technique.

The method presented by Mattos [32] uses meta-heuristic search techniques to generate test data dynamically. The approach evaluates two algorithms: the genetic and the bacteriological algorithm, and conclude that the latter generates better test data. The authors applied mutation testing with three mutation operators proposed for MapReduce-based programs to evaluate the work. They propose an operator that inserts an operation *combine* with the same behaviour as the *reduce*; an operator that removes the *combine*; and an operator that changes the number of processes that execute the *reduce*. These operators simulate semantic faults concerning the understanding of the MapReduce model. The conclusion is that the bacteriological algorithm performs well in generating test data

Prepared using stvrauth.cls
DOI: 10.1002/stvr

and contributes to identifying functional faults, but that it has not contributed to the identification of faults related to the misunderstanding of the MapReduce model.

Li *et al.* [33] introduce a *testing framework* for Big Data processing programs that extracts test data from real datasets. Their goal is to extract a subset of test data from the dataset itself, which will be processed using *input space partitioning techniques*. The *framework* gets a representative subset of data from the original dataset. This subset of test data is then applied in the validation process of the program under test.

Static testing of control flow (MapReduce) programs. Some works test MapReduce programs without considering the program's execution, using formal methods and static analysis. A work by Chen *et al.* [34] makes use of formal methods to verify the commutativity of *reduce* operations in MapReduce programs. Since this program is executed in a parallel and distributed computing environment, aggregation operations must be commutative and associative to ensure a deterministic result. The constraint about determinism is explained by the fact that it is impossible to determine the order in which the values will be processed [35]. The method presented in [34] extracts a series of assertions and properties from the *reduce* operation. These assertions must be verified through an external program, such as a *model checker*, to attest that the operation is commutative.

A method for static type analysis in MapReduce programs is proposed by Dörre *et al.* [36]. The authors aim to identify type incompatibilities in the emission of key/value pairs in *map*, *combine* and *reduce* operations. This static analysis is done automatically with the *SNITCH* tool (*StatIc Type Checking for Hadoop*) proposed by the authors. Ono *et al.* [37] verify the correction of MapReduce programs. This correction is done formally using *Coq* [38], an interactive theorem tester.

3.2. Data Flow Program Testing

Data flow programming is a paradigm that models a program as a directed graph of the data flowing between operations. It promotes the modelling of programs as a series of connections among operations. Explicitly defined inputs and outputs connect operations, which work like black boxes. Instructions do not impose any constraints on sequencing except for the data dependencies. An

Prepared using stvrauth.cls
DOI: 10.1002/stvr

operation runs as soon as all of its inputs become available. Thus, data flow languages are inherently parallel and can work well in large, decentralized systems.

Data flow programming has been adopted by libraries and environments addressing data analytics and processing like MatLab, R and Simulink. Big data parallel processing solutions like Apache Spark [5], DryadLINQ [3], Apache Beam [4] and Apache Flink [2], also adopt data flow programming models for implementing programs as processing data flows.

In general, testing a data flow program involves *(i)* defining testing criteria, *(ii)* classifying paths on the data flow graph that satisfy these criteria, and *(iii)* developing path predicate expressions to derive test input.

Testing data flow based data analytics programs. Data flow analytics has been promoted by Simulink, Matlab and R environments. Models implemented as functions to be used within programs in this context can also become complex to test. Therefore, approaches have been devoted to promoting full or partial testing strategies. For example, the model checking engine COVER [39] defines a verification methodology to assess the correctness of Simulink models. COVER automatically generates test cases and adopts fault and mutation-based testing. Therefore, coverage of a Simulink program by a test suite is defined in terms of detecting injected faults. The work can compute test suites for given fault models using bounded model checking techniques. MATmute [40] is based on an approach for automatically generating test suites for Scientific MATLAB code. The approach introduced by Xu *et al.* [41] improves the dependability of data flow programs by checking operators for necessary properties. The approach is dynamic and involves generating tests whose results are checked to determine whether specific properties hold or not.

Testing data flow based big data parallel programs. For testing *Apache Spark*, *DryadLINQ*, *Apache Beam*, and *Apache Flink* programs, it is possible to implement unit tests using external libraries and functions provided by these systems. Libraries such as those developed by Karau [42] and Otto Group [43] have adopted this strategy. They offer several utility classes for unit testing in Apache Spark and Apache Flink, respectively. For Apache Beam and DryadLINQ, it is possible to

Prepared using stvrauth.cls
DOI: 10.1002/stvr

use native support that enables the definition of program entries, automates the execution of tests and proposes *test oracle* to verifying program results. Although these libraries are essential for implementing and executing tests, they do not support test cases' design, which is a critical part of the testing process.

Regarding test case design, the work by Riesco *et al.* [44] applies *property-based testing* [45] to test Spark programs. This technique consists of generating random data, running the program under test with this input data, and then verifying the program's behavior through oracles that verify if the program's results meet specific defined properties through logical expressions. Riesco *et al.* [44] present *sscheck*, a library for the application of property-based testing in Spark programs. That work was extended by the same group ([46, 47]) to test stream processing programs with *Spark Streaming*. The authors applied temporal logic to verify time-related properties, which is an essential variable in stream processing. Their work was adapted for Apache Flink by Espinoza et al [48].

3.3. Discussion

Data processing programs operate on large amounts of data, making the data distribution across the system nodes significantly important. In classic parallel processing programs systems, parallelism is vital for avoiding processing bottlenecks. In contrast, in a data processing program, extensive input and output data streams make the network and the disk I/O the bottleneck rather than the CPU. Consequently, the use of parallelism is strongly determined by the communication across the nodes of the system. Also, parallelism is exploited to enhance parallel disk I/O. A consensus on parallel and distributed data processing system architectures has emerged, based on a so-called *shared nothing* hardware design. In a shared nothing system several nodes, each having its own processor, memory modules and secondary storage devices, are connected by a local area network (LAN). The only way processors communicate with one another is by sending messages via this interconnection network.

Parallel programs do not only include data processing operations (filtering, selection, clustering); they include data I/O directives used to guide control and data sharing across nodes. Testing a control

Prepared using stvrauth.cls
DOI: 10.1002/stvr

and data flow of a parallel processing program implies testing how data processing operations and data management operations are coordinated during the execution of a parallel program. In both cases, the code that implements the data processing operations using a target programming language does not introduce further challenges than classic programs testing. However, program testing must include the code used to deal with read/write operations, global/local variables used to share data in memory/cache, and exchange it across processes. In data flow-based programs, testing must consider the way data management operations are weaved in the program logic.

The works presented previously focusing on parallel programs using the MapReduce programming model do not provide tools for automatically testing programs, or they have not been widely experimented [23, 24]. The principle of testing data flow-based programs involves determining some programs' properties by analyzing or verifying properties on the data flow. The challenge consists of (automatically) generating tests and then developing tools for evaluating these tests on top of programs. The characteristics of the programming languages promoting a data flow programming model guide the type of generated tests that consider input and output data of operations applies on top of them. This absence of testing support shows that testing Big Data processing programs still lacks techniques and tools, mainly for testing programs that follow a data flow model.

4. MUTATION OPERATORS FOR APACHE SPARK BIG DATA PROCESSING PROGRAMS

Mutation testing is based on the definition of "mutation operators" which operate on programs, to simulate faults. Faults in a program are modeled beforehand according to the characteristics of the programming language. For example, faults can be related to a missing iteration in a loop or a mistake in an arithmetical or logical expression.

In a previous paper [10], we specified a set of mutation operators to mimic common faults and mistakes in data flow parallel Big Data processing programs (such as Apache Spark programs). These operators specify modifications in two levels:

Prepared using stvrauth.cls
DOI: 10.1002/stvr

1. the DAG that defines the program's *data flow*, for example, changing the calling order of two transformations;
2. inside *transformations* (see Section 2), such as changing the function passed as input to an aggregation transformation.

The operators were defined using our experience in the development of a taxonomy of functional faults in Apache Spark programs [49].

In this section we briefly describe a taxonomy of functional faults in Apache Spark programs [49] (Section 4.1), followed by sections defining mutation operators by each level.

4.1. A Taxonomy of Functional Faults in Apache Spark Programs

The design of big data processing Spark programs includes ensuring their intended functional behavior. Functional faults may result in unexpected behavior. Examples of these faults are the use of wrong transformations, incorrect parameters and absence of transformations.

A set functional faults that may appear in Apache Spark Big Data processing programs are classified in a taxonomy proposed by Souza Neto [49]. The taxonomy was proposed after the analysis of Apache Spark documentation, other references in the literature, as well as the analysis of the source code of Spark programs. The taxonomy classifies three main types of functional faults related to the *data flow*, the strategy and order in which *operations* are called, and the use of *accumulators*.

Faults related to the *data flow* of a Spark program refer to (i) an incorrect order in which operations are called; and (ii) use of the wrong operation. To illustrate these faults, let us consider the Spark code shown in Figure 4a that (correctly) analyzes log files to count the number of error messages. Under the hypothesis that error messages contain the initial string "ERROR", the code first filters all messages starting with the word "ERROR"; then, it removes the beginning of each message, by applying a map function. Finally, it filters messages containing the word "foo".

In contrast, Figure 4b shows a version of the code where the two filtering operations are called in the wrong order (inversion of lines 2 and 4) tagged with Δ_1 and Δ_3. By inverting the filters,

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
1  val fooCount = logsRDD
2    .filter(_.startsWith("ERROR"))
3    .map(_.split('\t')(2))
4    .filter(_.contains("foo"))
5    .count
```

(a) Correct Program

```
1  val fooCount = logsRDD
2  Δ1 .filter(_.contains("foo"))
3  Δ2 .flatMap(_.split('\t'))
4  Δ3 .filter(_.startsWith("ERROR"))
5    .count
```

(b) Faults in the Data Flow

```
1  val fooCount = logsRDD
2  Δ4 .filter(_.endsWith("ERROR"))
3  Δ5 .map(_.split(' ')(2))
4  Δ6 .filter(_.equals("foo"))
5    .count
```

(c) Faults in Operations

Figure 4. Example of a Spark program for log analysis.

the result is different: First, the program obtains a list of messages containing the word "foo", including messages beginning with "ERROR". Then, the function map removes the initial substrings of each message. Consequently, the string "ERROR" is deleted from messages that correspond to those intended to be retrieved. The final result is a list of messages that do not necessarily contain "ERROR" at the top or contain this string in their "body".

The line tagged with Δ_2 in the code illustrates the incorrect use of an operation in a data flow. The program statement uses the wrong type of map function since it calls *flatMap* instead of *map*. The operation *map* expresses a one-to-one transformation, transforming each element of a collection into one element of the resulting collection. In contrast, the operation *flatMap* expresses a one-to-many transformation, so it transforms each element of a collection to 0 or more elements of the resulting collection. The *map* operation applied on line 3 (Figure 4a) splits the log message using the separator "\t" and filters the third element from the resulting data (RDD). The second version of the code (Figure 4b) the operation *flatMap* (line 3) just splits the log message based on the separator "\t". The resulting data (RDD) contains all the strings containing this separator. Nothing is done with the third element in comparison to the first version. Both programs produce a result of the same data type (RDD[String]), but with different content.

The second type of fault identified in the taxonomy concern the way *operations* are used and weaved to process data in a program. These faults include the incorrect definition of mapping,

Prepared using stvrauth.cls
DOI: 10.1002/stvr

filtering and aggregation operations, like passing an incorrect processing function as a parameter to the transformation. For instance, consider the examples of faults shown in Figure 4c. An example of an incorrect definition of filtering transformations in the code is tagged with Δ_4 and Δ_6. The line tagged with Δ_4 verifies whether the end of the message string is the word "ERROR" instead of looking for that string at the beginning. The line tagged with Δ_6 checks whether the whole message corresponds to the string "foo" instead of just checking if it contains that word.

The line tagged with Δ_5 illustrates the incorrect definition of a mapping transformation with erroneous functions passed as parameters. In this case, the separator (" ") passed to the function *split* called within the function *map* is different from the separator used in the correct version ("\t"). Thus, the list of strings that will be generated by splitting the message using " " will be different from those generated using "\t". Thus, the selection in the mapping will produce different results in both programs since the function selects the third substring.

This type of faults can also come up when using the following operations: (i) Choosing incorrect binary operations (joins and set-like operations), for example calling the wrong type of join operation[§]; (ii) Choosing the wrong sorting operations, for example, choosing the wrong order option (ascending or descending); (iii) Incorrect treatment of duplicate data, like not removing duplicate data from a result when necessary.

The third group of faults in the taxonomy concern *Accumulators*, which are variables that are only operated by using associative operations, therefore, being efficiently supported in parallel programs. Accumulators can be used to implement counters (as in MapReduce) or sums [13]. Spark natively supports accumulators of numeric types, and programmers can add support for new types. In the example below, we create an accumulator variable (accum) and use it to aggregate values within a Spark operation:

```
var accum: LongAccumulator = sc.longAccumulator
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))
```

[§]Note that Spark proposes different implementations of the operator join with different semantics. The programmer must be sure of the type of join pertinent to the application logic.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Faults related to accumulators concern the commutative and associative properties of the operations on them. Indeed, accumulators in Spark can be commutative and associative and the right type must be chosen in the code depending on the way data should be shared across processes to aggregate partial results. The programmer must have a clear understanding of the properties of the task. The example below illustrates a mistake made when using a standard variable instead of an accumulator. Spark handles variables and accumulators differently, a variable will be copied and managed locally on each node in the cluster, so the final result will not contain the total sum produced in parallel by all the nodes. In contrast with an accumulator, the final result will aggregate the partial sums computed by each node.

```
var accum: Long = 0
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
```

Through our study, we identified 16 possible faults that can appear in Spark programs. The reader may refer to the first author's thesis [49] for details on this study. In the next sections we define our mutation operators [10], namely mutation operators for: *data flow* and *transformations*.

4.2. Mutation Operators for Data Flow

The operators defined in this group replace, swap and delete unary and binary transformations calls. Recall that a unary transformation operates on a single input RDD and a binary transformation on two input RDDs.

Unary Transformations Replacement (UTR) Replaces a unary transformation for another with the same input and output types.

Unary Transformation Swap (UTS) Swaps the calling order of two unary transformations in the program, provided that they have the same input and output signature.

Unary Transformation Deletion (UTD) Removes the call of a unary transformation that receives and returns RDDs (both) of the same type in the program.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

The operators **Binary Transformation Swap (BTS)** and **Binary Transformations Replacement (BTR)** are similar to their unary versions, but operate on binary transformations.

Operators in this group change the DAG that defines the data flow of a Spark program. To illustrate these mutation operators, let us consider the program shown in Figure 4a that filters error messages in a log. Table I shows examples of the mutants that can be generated for this program applying the data flow mutation operators. Mutant 1 was generated by the operator UTR, replacing the filtering transformation in line 2 with the mapping transformation in line 3 and no changes in lines 3 and 4. Mutant 2 was generated by the operator UTS swapping lines 2 and 3. Mutant 3 was generated by the operator UTD deleting the filtering transformation from line 4.

Table I. Examples of mutants generated with the data flow mutation operators.

Id	Operator	Lines	Mutation
1	UTR	2	**val** fooCount = logsRDD .map(_.split('\t')(2)) .map(_.split('\t')(2)) .filter(_.contains("foo")) .count
2	UTS	2, 3	**val** fooCount = logsRDD .map(_.split('\t')(2)) .filter(_.startsWith("ERROR")) .filter(_.contains("foo")) .count
3	UTD	4	**val** fooCount = logsRDD .filter(_.startsWith("ERROR")) .map(_.split('\t')(2)) .count

4.3. Mutation Operators for Transformations

The operators of this group replace, invert, insert and delete specific transformations in a Spark program: mapping, filtering, set-like, distinct, aggregation, join and order. Table II shows examples of the mutation operators of this group using excerpts from the Spark programs shown in Figure 2 and Figure 4a. We also use other code snippets to illustrate mutation operators that modify transformations that are not applied in previous programs' examples. We also define reductions rules that define the conditions in which the application of some operators override the application of others.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table II. Examples of use of the transformation mutation operators.

Id	Operator	Fig.	Line	Mutation
1	MTR	2	2	**val** words =input.flatMap((line: String)=>...originalValue.headOption)
2	MTR	2	2	**val** words =input.flatMap((line: String)=>...originalValue.tail)
3	MTR	2	2	**val** words =input.flatMap((line: String)=>...originalValue.reverse)
4	MTR	2	2	**val** words =input.flatMap((line: String)=>...Nil)
5	NFTP	4a	4	... .filter(!_.contains("foo"))
6	STR	–	–	**val** rdd3 =rdd1.union(rdd2)
7	STR	–	–	**val** rdd3 =rdd1.intersection(rdd2)
8	STR	–	–	**val** rdd3 =rdd1
9	STR	–	–	**val** rdd3 =rdd2
10	STR	–	–	**val** rdd3 =rdd2.subtract(rdd1)
11	DTD	–	–	**val** rdd4 =rdd3
12	DTI	2	2	**val** words =input.flatMap((line: String)=>line.split("")).distinct()
13	DTI	2	3	**val** pairs =words.map((word: String)=>(word, 1)).distinct()
14	DTI	2	4	**val** counts =pairs.reduceByKey((a: Int, b: Int)=>a + b)).distinct()
15	ATR	2	4	**val** counts =pairs.reduceByKey((a: Int, b: Int)=>a))
16	ATR	2	4	**val** counts =pairs.reduceByKey((a: Int, b: Int)=>b))
17	ATR	2	4	**val** counts =pairs.reduceByKey((a: Int, b: Int)=>a + a)
18	ATR	2	4	**val** counts =pairs.reduceByKey((a: Int, b: Int)=>b + b)
19	ATR	2	4	**val** counts =pairs.reduceByKey((a: Int, b: Int)=>b + a)
20	JTR	–	–	**val** rdd4 = rdd3.leftOuterJoin(rdd2) .map(x => (x._1, (x._2._1, x._2._2.getOrElse("")))))
21	JTR	–	–	**val** rdd4 = rdd3.rightOuterJoin(rdd2) .map(x => (x._1, (x._2._1.getOrElse(0), x._2._2)))
22	JTR	–	–	**val** rdd4 = rdd3.fullOuterJoin(rdd2) .map(x => (x._1, (x._2._1.getOrElse(0), x._2._2.getOrElse(""))))
23	OTD	–	–	**val** rdd2 =rdd1
24	OTI	–	–	**val** rdd2 =rdd1.sortByKey(ascending =**false**)

Mapping Transformation Replacement (MTR) Given a mapping transformation (*map*, *flatMap*) that receives a mapping function f as parameter, the operator MTR replaces f by a mapping function f_m, where *(a)* f_m returns a constant value of the same type as f, or *(b)* f_m modifies the value returned by f. For example, for a mapping function that operates on an integer parameter, MTR defines five cases where f_m defines the following value types to be returned: the constants 0, 1, Max and Min (these two denote the largest and lowest values of the integer type), and the original output value of f but with an inverted sign.

Table III shows mapping values of basic types and collections that we defined as output parameter types for f_m. In the table, x represents the value generated by the original mapping function; k and v represent the key and value generated by the original mapping function in the case of key/value

Prepared using stvrauth.cls
DOI: 10.1002/stvr

tuples; k_m and v_m represent modified values for the key and value, that correspond to the application of other mapping values respecting the type.

Table III. Mapping values for basic and collections types.

Type	Mapping Value
Numeric	$0, 1, MAX, MIN, -x$
Boolean	$true, false, \neg x$
String	" "
List	$List(x.head)$, $x.tail$, $x.reverse$, Nil
Tuple	(k_m, v) , (k, v_m)
General	*null*

To illustrate the MTR operator, consider the mapping transformation applied to line 2 of the words count program in Figure 2: **val** words =input.flatMap((line: String)=>line.split("")).

From this line, the MTR operator generates mutants 1–4 in Table II. For the sake of simplicity, we are only showing the modified line, hiding some details, such as calling the original function to obtain the original value we need in some mutants (*originalValue* refers to this original value).

Filter Transformation Deletion (FTD) this operator creates a mutant for each *filter* transformation call in a given program, deleting one *filter* at a time.

Reduction rule: This operator is a specific case of UTD (unary transformation deletion) because it generates mutants where filters that are a type of unary transformations have been deleted. So during a mutation process, FTD is applied if UTD has not been applied before.

Mutant 3, shown in Table I illustrates mutant 3 generated with the operator UTD. The operator FTD applied to the filtering transformation in line 4 of Figure 4a generates this mutant.

Negation of Filter Transformation Predicate (NFTP) Given a *filter* transformation call with a predicate function p as input parameter, the operator NFTP replaces the predicate function p with a predicate function p_m that negates the result of the original function ($p_m(x) = \neg p(x)$). As an example of this operator, consider the same filtering transformation used to illustrate the operator FTD. The operator NFTP in this transformation generates mutant 5 in Table II.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Reduction rule: The application of this operator is overridden by applying the operators UTD or FTD. Even if the mutants generated by the operator NFTP are not generated by the operators UTD or FTD, we experimentally observed that tests designed to kill the mutants that delete unary/filtering transformations kill the mutants that negated the filtering predicate (as those generated by NFTP).

Set Transformation Replacement (STR) For each occurrence of a set transformation (*union*, *intersection* and *subtract*) in a program, this operator creates five mutants: (1-2) replacing the transformation by each of the other remaining set transformations; (3) keeping just the first RDD; (4) keeping just the second RDD; and (5) changing the order of the RDDs in the transformation call (only for *subtract*, since *union* and *intersection* are commutative). For example, given the following excerpt of code with a *subtract* between two RDDs:

```
val rdd3 = rdd1.subtract(rdd2)
```

The operator STR applied on this transformation creates the five mutants, described at lines 6–10 in Table II.

Distinct Transformation Deletion (DTD) For each call of a *distinct* transformation in the program, this operator creates a mutant by deleting its call. As the *distinct* transformation removes duplicated data from the RDD, this mutation keeps the duplicates. For example, the application of DTD in the following excerpt of code generates the mutant 11 of Table II:

```
val rdd4 = rdd3.distinct()
```

Reduction rule: The DTD operator is a specific case of the operator UTD. It generates mutants that are also generated by the operator UTD. So DTD is applied if UTD has not been applied before.

Distinct Transformation Insertion (DTI) For each transformation (other than *distinct*) in the program, this operator creates a mutant inserting a *distinct* transformation call after that transformation. Applying DTI to the transformations presented in Figure 2 generates the mutants 12–14 of Table II.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Aggregation Transformation Replacement (ATR) Given an aggregation transformation with an aggregation function f as input parameter the operator ATR replaces f by a different aggregation function f_m. The definition of ATR considers five replacement functions. Given an original function $f(x, y)$, the corresponding replacement functions $f_m(x, y)$ are: (1) a function that returns the first parameter ($f_m(x, y) = x$); (2) a function that returns the second parameter ($f_m(x, y) = y$); (3) a function that ignores the second parameter and calls the original function with a duplicated first parameter ($f_m(x, y) = f(x, x)$); (4) a function that ignores the first parameter and calls the original function with a duplicated second parameter ($f_m(x, y) = f(y, y)$); and (5) a function that swaps the order of the parameters ($f_m(x, y) = f(y, x)$), which generates a different value for non-commutative functions. Table II shows the mutants 15–19 as examples of mutants generated by the operator ATR applied to the aggregation transformation on line 4 in Figure 2 (see this excerpt below).

```
val counts = pairs.reduceByKey( (a: Int, b: Int) => a + b) )
```

Join Transformation Replacement (JTR) For each occurrence of a join transformation ((inner) *join*, *leftOuterJoin*, *rightOuterJoin* and *fullOuterJoin*) in a program, the operator JTR replaces that transformation by the three remaining join transformations. Additionally, a map transformation is inserted after the new join to adjust it, typing with the replaced one. This adjustment is necessary to maintain the type consistency between the mutant and the original program. Indeed depending on the join type, the left, right, or both sides can be optional, and the resulting RDD can be slightly different. For example, replacing the transformation (inner) *join* by *rightOuterJoin* makes left-side values optional. To keep type consistency with the original transformation, we map empty left-side values to default values, in case of basic types, or *null*, otherwise.

To illustrate the operator JTR, consider the following code snippet that joins two RDDs:

```
val rdd4 = rdd3.join(rdd2)
```

Assuming that rdd2 and rdd3 are of types RDD[(Int, String)] and RDD[(Int, Int)], respectively, the rdd4 is of type RDD[(Int, (Int, String))]. Applying JTR to this transformation generates the mutants 20–22 of Table II. Taking mutant 21 as an example, replacing *join* with *rightOuterJoin*, the resulting RDD is

Prepared using stvrauth.cls
DOI: 10.1002/stvr

of type RDD[(Int, (Option[Int], String))]. Thus, the *map* following the *rightOuterJoin* serves to set the value of type Option[Int] to Int. When this value is empty (*None*), we assign the value zero (0).

Order Transformation Deletion (OTD) For each order transformation (*sortBy* and *sortByKey*) in the program, this operator creates a mutant by deleting the transformation. From the code snippet **val** rdd2 =rdd1.sortByKey(), the operator OTD generates mutant 23 in Table II.

Reduction rule: The operator OTD is a specific case of UTD, deleting unary transformations of a specific type (order transformations). So, the application of UTD overrides the application of OTD.

Order Transformation Inversion (OTI) For each order transformation in a program, the operator OTI creates a mutant where the ordering (ascending or descending) is replaced by the inverse one. Considering the same code snippet that was used as an example for the OTD operator, the application of the OTI operator in this transformation generates mutant 22 in Table II, where the ascending ordering that is true by default was changed for false.

Reduction rule: The operators UTD and OTD override the operator OTI. Even if the operator OTI generates different mutants as those generated by UTD and OTD, we experimentally observed tests designed to kill the mutants that delete an order transformation (as those generated with UTD or OTD) also kill the mutants generated by OTI that invert the order.

Figure 5 depicts the Apache Spark functional faults taxonomy presented in Section 4.1 (left side) and the mutation operators presented in sections 4.2 and 4.3 (right side). In the figure we can also see the relationship between them to see what types of fault each operator can simulate.

5. TRANSMUT-SPARK

This section presents TRANSMUT-SPARK (*Transformation Mutation for Apache Spark*), a tool that automates the mutation testing process of Spark programs written in Scala. The tool was developed as a *plugin* for *SBT* (*Scala Build Tool*) [50], a tool for building projects in Scala and Java. TRANSMUT-SPARK automates the process of generating the mutants, applying the mutation

Prepared using stvrauth.cls
DOI: 10.1002/stvr

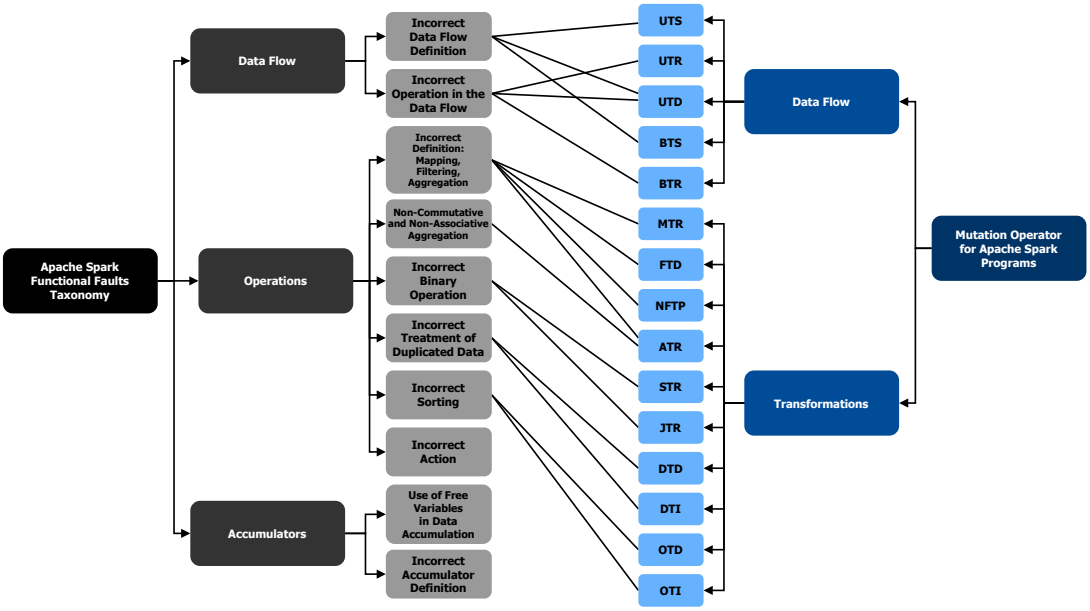


Figure 5. Taxonomy of Apache Spark Functional Faults and corresponding Mutation Operators [49, 10].

operators presented in Section 4, executing the tests on the original program and the mutants, and analyzing the test results, generating a report with metrics and process results.

5.1. Mutation Process Workflow

TRANSMUT-SPARK automates the main steps of the mutation testing process, which includes the processes of mutants generation and execution of the tests with the mutants. The main functionalities implemented by the tool are: program analysis, mutant generation, test execution and mutant analysis.

Figure 6 shows an overview of the workflow implemented by TRANSMUT-SPARK. The figure shows the tool's main modules with their input and output and the flow among modules.

Program analysis: TRANSMUT-SPARK receives as input a source code containing the Spark program under test *(a)*. This code is analyzed by **`ProgramBuilder`** module *(b)* to identify the principal elements and places necessary to apply the mutation operators. From this analysis, the module generates an intermediate representation of the program *(c)* using an abstract model for data flow programs [51]. In this model, a program is defined from a set of *datasets* (RDDs), a set of *transformations* and a set of *edges* that make the connection between the datasets and

Prepared using stvrauth.cls
DOI: 10.1002/stvr

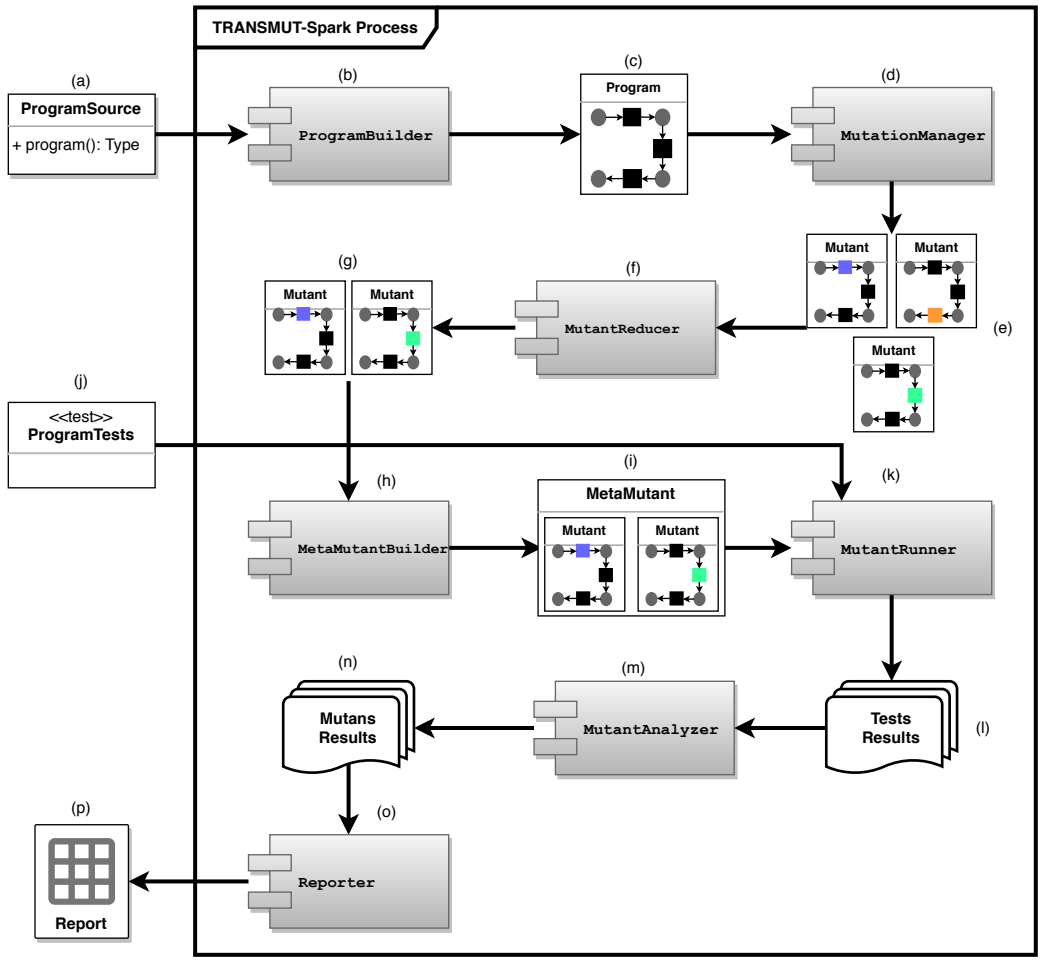


Figure 6. Overview of the workflow of TRANSMUT-SPARK.

transformations, forming the data flow of the program, in the form of an Directed Acyclic Graph (DAG).

Mutant generation: TRANSMUT-SPARK generates mutants for the Spark program under test by applying the mutation operators presented in Section 4. The generated mutants are incorporated in a single source code, called *meta-mutant*, in order to reduce the amount of code to be compiled and managed. The task of generating the mutants is carried out by three of TRANSMUT-SPARK modules.

First, the intermediate representation is passed to the **`MutationManager`** module *(d)*, responsible for applying the mutation operators presented in Section 4 and generating a set of

Prepared using stvrauth.cls
DOI: 10.1002/stvr

mutants *(e)*. For each mutation operator, the module checks whether it applies to the set of transformations. If so, it generates all the mutants that can be generated from this operator.

The next steps consist of generating executable versions of the mutants and executing them with the test set to analyze the results. As mentioned in Section 2.2, mutation testing can have a high computational cost due to the extensive amount of mutants that can be generated and executed in the process. Because of that, different works have proposed techniques to reduce the costs of mutation testing [52]. In TRANSMUT-SPARK we apply two techniques for cost reduction: *selective mutation* [53], and *Mutant Schema Generation* (MSG) [54].

Selective mutation reduces the number of mutants that will be executed by removing redundant, equivalent, or trivial mutants (which are easily killed). In TRANSMUT-SPARK, this technique is applied by **`MutantReducer`** *(f)*. This module takes the set of mutants *(e)* and applies *reduction rules* to remove redundant mutants from the set. These rules were stated in Section 4 and they are summarized in Table IV.

The reduction rules R_1, R_2 and R_3 were established given the redundancy of the operators FTD, DTD and OTD with the UTD operator which represents a general transformation deletion, as well as the subsumption relationships between FTD and NFTP, and between OTD and OTI operators, as discussed in Section 4. The reduction rules R_4, R_5 and R_6 result from the analysis of the experimental results described in Section 6. These results empirically showed that some of the generated mutants do not lead to the design of new test cases. If the number of mutants had to be reduced, these mutants would be good candidates to be removed. Note that the use of the module **`MutantReducer`** is optional and configurable, so a user can select which reduction rules to apply according to the needs of his/her project.

The module **`MutantReducer`** generates as output a new set of mutants *(g)*, which is a subset of the original without redundant mutants. On the next step, the set of mutants *(g)* is passed to the module **`MetaMutantBuilder`** *(h)*. This module implements MSG, the second cost reduction technique applied in TRANSMUT-SPARK. In this technique, all the mutants are incorporated into a single source program code, generating a "*meta-mutant*" that incorporates all the mutants

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table IV. Reduction rules.

Rule	Description
R_1	Removes mutants generated with the mutation operators FTD, DTD and OTD when the UTD operator has also been applied.
R_2	Removes mutants generated with the mutation operator NFTP when the FTD or UTD operators have also been applied.
R_3	Removes mutants generated with the mutation operator OTI when the OTD or UTD operators have also been applied.
R_4	Removes the following mutants generated with the MTR operator: mutants that map to *Max* and *Min*, when the mapping is to a numerical type; mutants that map to " ", when the mapping is to the type string; mutants that map to $x.reverse$, when the mapping is to a collection type; and mutants that map to $null$, when the mapping is to any other type.
R_5	Removes mutants generated with the mutation operator DTI when the *distinct* transformation has been inserted after grouping or aggregation transformations.
R_6	Removes the commutative replacement mutants ($f_m(x, y) = f(y, x)$) generated with the ATR mutation operator.

generated individually but that only needs to be compiled once. The MSG technique was used in TRANSMUT-SPARK because it is faster than other techniques [55], such as the interpretation-based technique that is used by classical mutation testing tools, such as *Mothra* [56], or the *separate compilation* approach that is used by *Proteum* [12]. Thus, the **`MetaMutantBuilder`** module *(h)* receives the mutant set *(g)* as input, aggregates all mutants into a single program, and generates a meta-mutant as output *(i)*.

Test execution: Then, the meta-mutant and the class that implements the tests *(j)* are passed to the module **`MutantRunner`** *(k)*. This module is responsible for managing the execution of the tests with the original program and mutants. TRANSMUT-SPARK first executes the tests with the original program and checks if their results are as expected. If the original program tests fail, the tool ends the process and indicates that the original program or the tests need to be fixed. Otherwise, the tool executes the test set for each mutant and stores its results *(l)*.

Mutant analysis: These results are then passed to the **`MutantAnalyzer`** module *(m)*, which is responsible for analyzing the results of each mutant, checking whether the tests passed or failed, and returning the status (killed or lived) of each mutant *(n)*. Finally, the results of the analysis are passed to the module **`Reporter`** *(o)* which is responsible for computing metrics, such as the number of

Prepared using stvrauth.cls
DOI: 10.1002/stvr

mutants, the number of killed mutants, and mutation score, generating reports with the results of the process *(p)*.

5.2. Implementation details and architecture

The TRANSMUT-SPARK tool is implemented in *Scala* [57]. We also chose Scala as the working language supported by the tool. Scala is a high-level language that incorporates object-oriented and functional programming; it is statically typed and executed in the JVM. Scala also is the most used programming language for Spark, offering better interaction with programs in that language since Spark was also developed in Scala [5].

TRANSMUT-SPARK was developed as a plugin for *SBT* (*Scala Build Tool*) [50], a tool for building, managing, and deploying software projects in Scala and Java. SBT provides an interactive command-line interface that automates different software development tasks, including the compilation, tests, and publication of the project. SBT has the advantage of providing a simplified configuration language, increase productivity with automated build and test tasks, and can be easily extended through plugins [50]. Developing the tool as a plugin allows direct access to the compiler and test execution components of SBT necessary to run the mutation testing process. Lastly, a plugin can be easily added to any project that uses SBT as a build tool, not requiring any additional installation, environment preparation, or specific operational system, making the tool portable and quickly adopted by different projects.

Figure 7 presents an overview of the architecture of TRANSMUT-SPARK. The project is divided into six sub-projects that implement modules responsible for different mutation testing processes. Each sub-project is presented below:

`util`: contains utility classes that are used by all other modules;

`code-analyzer`: analyzes the source code of a Spark program, doing a syntactic and type analysis. It generates an intermediate representation of the program;

`mutation-manager`: contains the classes that implement the mutation operators presented in Section 4 and implements the modules for generating mutants, reducing mutants, implementing the

Prepared using stvrauth.cls
DOI: 10.1002/stvr

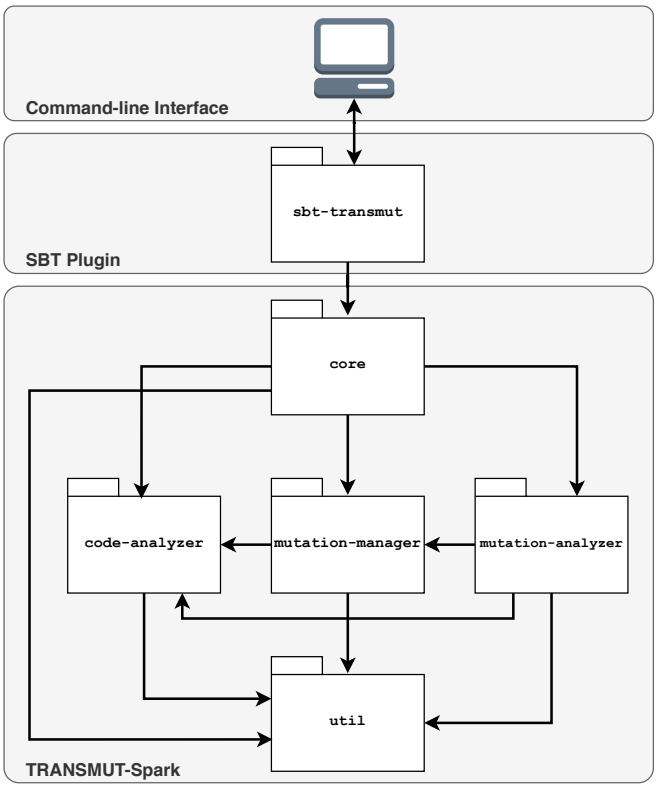


Figure 7. Overview of the TRANSMUT-SPARK architecture.

reduction rules, and building the meta-mutant that aggregates all the mutants generated into a single source code (modules *(d)*, *(f)* and *(h)* in Figure 6);

`mutation-analyzer`: contains the interface to the module that manages the execution of the tests with the mutants and the original program, and the module that analyzes the results of the tests, indicating if the mutants were killed in the mutation testing process (modules *(k)*, *(m)* and *(o)* in Figure 6);

`core`: main project that aggregates the other modules, implements the tool settings and defines the interface for the module that implements the mutation testing process. This module operates as orchestrator for the workflow, to execute the mutation testing process presented in Figure 6;

`sbt-transmut`: implements the TRANSMUT-SPARK SBT plugin. This plugin defines the tasks that can be executed through the command-line interface of SBT. In addition, this project implements the concrete modules of the mutation testing process and test execution, dependent on the compiler and test components of SBT.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

For the development of the module **`ProgramBuilder`**, that belongs to **`code-analyzer`** and does the syntactic and type analysis of the program code to be tested, we used the *Scalameta*[¶], a library for reading, analyzing, and transforming Scala code. This library was also used to implement the mutation operators of **`mutation-manager`** to make the changes in the source code.

For the development of the SBT plugin (**`sbt-transmut`**), we take as reference the SBT plugin of the tool Stryker [58], an open-source tool for traditional mutation testing of programs in JavaScript, C#, and Scala. We used it because it implements functionalities similar to the ones needed for our tool. Other modules of TRANSMUT-SPARK have been implemented differently from the modules of Stryker.

For the development of the module **`MetaMutantBuilder`**, that belongs to **`mutation-manager`** and receives a set of mutants and returns the meta-mutant, we used the technique of *mutation switching* that is also applied by Stryker [58]. This technique consists of putting all the mutants inside conditional expressions in the code and activating the mutant that must be executed through an environment variable. The component that controls the execution of the mutants is in charge of assigning values to that variable to indicate which mutant must be executed.

The TRANSMUT-SPARK project has approximately 16k lines of code in Scala, including 10k lines of code of tests. TRANSMUT-SPARK is an open-source project distributed under the MIT License. Details on the use of the tool and its source code can be found in the repository: `https://github.com/jbsneto-ppgsc-ufrn/transmut-spark`.

5.3. Use of the tool

The following configuration steps must be done for executing TRANSMUT-SPARK. First, add it as a plugin in a project that uses SBT and create a configuration file `transmut.conf` at the project's root. The file must contain at least the file names of the source code of the Spark program that will be transformed (sources), and the names of the methods that encapsulate the program (programs). The

[¶] `https://scalameta.org`.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
1  transmut {
2    sources: [ "WordCount.scala" ],
3    programs: [ "wordCount" ],
4    test-only: [ "example.WordCountTest" ]
5  }
```

Figure 8. Example of a TRANSMUT-SPARK configuration file.

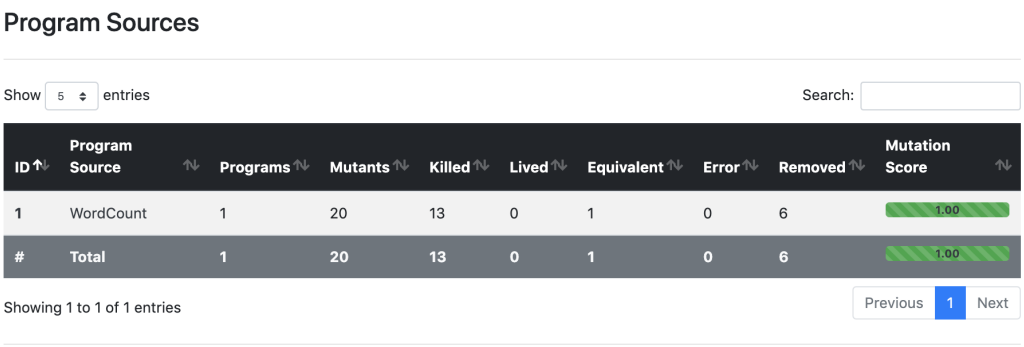


Figure 9. Part of the HTML report generated by TRANSMUT-SPARK with metrics about the programs.

Spark program that will go through mutation testing must be encapsulated in a method, so only that method is modified. Additional settings can be added to the file, such as a list of mutation operators (the tool applies all by default), test classes and reduction rules that will be applied. Figure 8 shows a configuration example for the program in Figure 2.

To execute TRANSMUT-SPARK, execute the command `sbt transmut` from the project folder in the command-line terminal. This command triggers the execution of the mutation testing process following the workflow presented in Figure 6. When the execution terminates, TRANSMUT-SPARK generates reports with process results (HTML and JSON documents). The reports include the information necessary to complete the mutation testing process. Figures 9, 10, 11 and 12 present part of the HTML report generated by TRANSMUT-SPARK for the program presented in Figure 2. These report includes the metrics about the program (Figure 9); information about the generated mutants (Figure 10); details of a mutant, such as its status and the code of the original program and the mutant (Figure 11); and general metrics about the mutation operators (Figure 12).

The reports show the mutants that are alive, and they can be analyzed to verify if they are equivalent. Identifying equivalent mutants allows verifying if the mutation testing process must

Prepared using stvrauth.cls
DOI: 10.1002/stvr

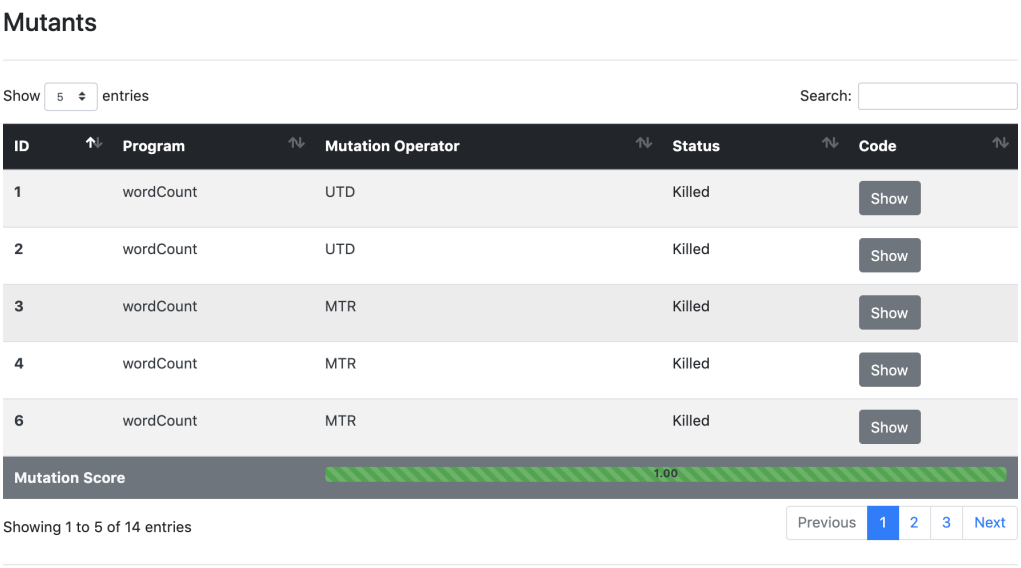
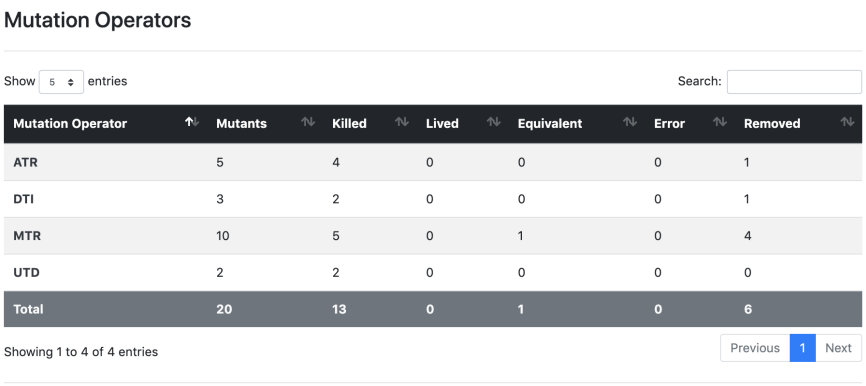


Figure 10. Part of the HTML report generated by TRANSMUT-SPARK with information about the generated mutants.

Figure 11. Part of the HTML report generated by TRANSMUT-SPARK with details of a mutant.

continue to achieve the established mutation score. To support the whole process, TRANSMUT-SPARK has the command `transmutAlive` that causes the process to be executed again only for the mutants that are alive of the last execution—this command tags in the report the mutants that have been identified as equivalent. The identification of equivalent mutants is made by inserting their

Prepared using stvrauth.cls
DOI: 10.1002/stvr



Mutation Operators

Show 5 entries Search:

Mutation Operator	Mutants	Killed	Lived	Equivalent	Error	Removed
ATR	5	4	0	0	0	1
DTI	3	2	0	0	0	1
MTR	10	5	0	1	0	4
UTD	2	2	0	0	0	0
Total	20	13	0	1	0	6

Showing 1 to 4 of 4 entries Previous 1 Next

Figure 12. Part of the HTML report generated by TRANSMUT-SPARK with metrics from the mutation operators.

identifiers in the tool settings in the field equivalent–mutants. This command forces the execution of mutants identified as relevant but removed by the reduction module. It also allows new tests to be inserted into the test class to kill non-equivalent mutants that are still alive. When using this command, a new report is generated with the updated metrics and results. More details on the use of the tool and its settings can be found in its repository.

6. EXPERIMENTAL SETUP

This section presents the experiments we conducted to evaluate TRANSMUT-SPARK. These experiments are complementary to those presented in our previous paper [10] that validated the transformation mutation approach.

6.1. Methodology

The experimental validation was designed to answer the following research questions about TRANSMUT-SPARK:

RQ_1 *Is* TRANSMUT-SPARK *applicable under reasonable costs to fully automate the mutation steps of the mutation testing process?*

The strategy is to analyze the applicability and the costs of using TRANSMUT-SPARK in Spark programs' mutation testing process in contrast to the manual experiments [10].

Prepared using stvrauth.cls DOI: 10.1002/stvr

RQ_2 *Which is the impact of activating the mutants reduction module in* TRANSMUT-SPARK *performance?*

The strategy is to compare TRANSMUT-SPARK performance when reduction rules are applied for reducing the number of mutants.

RQ_3 *To which extent* TRANSMUT-SPARK *and existing Scala programs mutation testing tools are complementary.*

The strategy is to compare experimentally TRANSMUT-SPARK with existing mutation tools and further assess the relevance of TRANSMUT-SPARK's underlying approach.

The set of experiments to answer these questions was conducted, adopting a methodology to measure the performance and interest of TRANSMUT-SPARK for automatizing testing and to compare it with existing tools.

For answering RQ_1 - RQ_3 we used TRANSMUT-SPARK to apply mutation testing, as described in Figure 3, to a set of nine representative Spark programs. For eight of the programs in the testing battery, we used the tests developed for manual experiments presented in a previous version of this research [10]. All tests were designed to kill non-equivalent mutants, aiming at a mutation score (*ms*) of 1.0. RQ_1 was tackled by the comparison between this previous experiment and the ones carried out with TRANSMUT-SPARK. For RQ_2 we activated the use of reduction rules to compare results.

To answer to RQ_3, we used *Scalamu*[‖], a traditional mutation testing tool to compare its results with those obtained by TRANSMUT-SPARK. *Scalamu* tests Scala programs. The tool is based on *PIT* [59], a state of the art tool for mutation testing of Java programs.

This experiment was designed to evaluate the performance of the (1) test set designed to kill the mutants generated by TRANSMUT-SPARK in the mutation testing process with *Scalamu*; (2) test set developed to kill the mutants generated by *Scalamu* in the process with TRANSMUT-SPARK.

[‖] https://github.com/sugakandrey/scalamu

Prepared using stvrauth.cls
DOI: 10.1002/stvr

The idea is to show that, since both tools (and mutation approaches) target different facets of the program, they may lead to tests that can identify different faults.

6.2. Experimental process

The mutation process with TRANSMUT-SPARK was executed in all of the testing batteries, alternatively enabling and disabling the reduction module and then to evaluate the mutation score obtained by the test set designed to kill *Scalamu* mutants. These experiments were performed on a MacBook Pro with a 1,4 GHz Intel Core i5 processor, 16 GB of RAM (2133 MHz LPDDR3) and a 256 GB SSD.

Assessment metrics We adopted the following metrics for comparing the three experiments:

1. time spent by the test developer in the process;
2. number of mutants and equivalent mutants;
3. execution time of the tool for testing experiment cases;
4. *Killed Ratio* (*KR*) to assess each mutation operator.

KR gives insights about the extent to which mutation operators tend to generate mutants challenging to kill, *i.e.*, that simulate faults that would be more difficult to detect. For a set of tests developed to achieve the mutation score of 1.0, *KR* corresponds to the ratio between the number of tests that killed the generated mutants and the total number of tests executed with those mutants. Low KR values show that mutants generated by a specific operator were killed by fewer tests, meaning they were hard to kill. In contrast, high KR values indicate that mutants were easily killed.

Furthermore, a qualitative comparison criterion used was to observe the occurrence of errors in the process.

Testing battery The testing battery used in the experiments consists of programs implementing representative types of Big Data processing tasks such as analysis of texts and log files, queries in

Prepared using stvrauth.cls
DOI: 10.1002/stvr

tabular databases, and data recommendation using the *collaborative filtering* algorithm [60]. The following lines briefly describe the programs applied in the evaluation**.

- *Exploring tabular datasets:* The programs `ScanQuery`, `AggregationQuery`, `JoinQuery` and `DistinctUserVisitsPerPage` explore tabular datasets containing websites data such as their ranking and visiting users. They implement the queries introduced in the *AMPLab Big Data Benchmark* [61], a benchmark for large data processing systems.
- *Analysing textual datasets:* The programs `NGramsCount` and `NasaApacheWebLogsAnalysis` analyse text datasets computing the frequency of n-grams and analysing log messages to identify unique messages in different files.
- *Analysing data thorugh programs with "complex" architectures (programs and sub-programs):* The programs `MoviesRatingsAverage`, `MoviesRecomendation` and `MovieLensExploration` analyse datasets produced by the application *MovieLens* [62]. It is a system where users can rate movies and receive recommendations based on their preferences. The program `MovieLensExploration`, composed of eight subprograms, performs a series of exploratory analyzes on the datasets of *MovieLens*.

The last program in the testing battery (`MovieLensExploration`) has a complex architecture that leads to many mutants, making it unsuitable for manual mutation testing. Because of that, this program was not applied in out previous experiments [10] where the mutation testing process was applied manually. As we did for the other programs in the previous experiment, tests for `MovieLensExploration` were incrementally developed to achieve an *ms* of 1.0.

Test case design To illustrate our test development strategy, let us consider the word count program shown in Figure 2. An example of a simple test for this program is an input data set containing a single word string. The expected result for this input dataset is the number of times that word appears, that is, only once. This simple test can kill mutants generated with the MTR

**The programs used in the experiments of this work are publicly available at https://github.com/jbsneto-ppgsc-ufrn/spark-mutation-testing-experiments.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

(mutation transformation replacement) operator that modifies the mapping transformations applied in the program. However, the fact that this test only contains a single word makes it unable to kill the mutants generated with the operators ATR (aggregation transformation replacement) and DTI (distinct transformation insertion). We created a new test containing an input dataset with more and repeated words to kill the mutants generated with these operators. The modifications of these operators can be detected. So we created this new test to kill the specific mutants that stayed alive with the first test. We terminated the process without creating more tests when no mutants were alive. This process was followed in all the experiments so that each program's test set had only the tests necessary to kill all the mutants generated by the tool.

For the comparison between *Scalamu* and TRANSMUT-SPARK we designed a new set of tests prepared to kill all mutations generated by *Scalamu* (i.e., ms of 1.0). We executed the two test sets with each testing tool *Scalamu* and TRANSMUT-SPARK (using the reduction module) and compared results. The programs used in this experiment were the same nine programs described above.

Experiments For each program in the testing battery we performed the following experiments:

E_1 (First Experiment): Manual mutation testing process. In this experiment, we applied the mutation testing process manually to have a baseline of the evaluation of the mutation operators [10].

E_2 (Second Experiment): TRANSMUT-SPARK, no reduction rules. In this experiment, we applied the mutation testing process to evaluate the tool and compare its results with the manual process. In this case, the reduction rules implemented in the tool were not applied.

E_3 (Third Experiment): TRANSMUT-SPARK with reduction rules. This experiment tests the effects of applying the reduction rules to the battery of tests of Experiment E_2.

E_4 (TRANSMUT-SPARK vs. *Scalamu*): In this experiment, we analyze and compare the behavior of both tools by cross-applying the tests generated by one tool to the other. The objective is to determine to which extent they are complementary.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Dealing with complex program structures (programs and subprograms) Applying the transformation mutation approach to programs composed of subprograms (methods) requires some strategic choices. In our previous manual experiment [10], transformations for generating mutants considered subprograms as a single program. Thus, mutation operations were applied considering the set of subprograms' as a whole, and so, transformations in two different subprograms were exchanged or replaced using the mutation operators. TRANSMUT-SPARK, in contrast, treats each subprogram as an independent program, with its own set of transformations. Thus, each subprogram is considered the transformations' scope of visibility considered by mutation operators. This choice had an impact on the number of generated mutants for each program.

7. RESULTS AND ANALYSIS

In this section, we present the results of our experiments, followed by a discussion to answer the research questions proposed to guide the experiments.

7.1. RQ1: Applicability of TRANSMUT-SPARK *to fully automate the mutation testing process steps*

The strategy adopted to answer RQ_1 was to analyze the applicability and the costs of using TRANSMUT-SPARK in Spark programs' mutation testing process in contrast to the manual experiments proposed in [10].

Table V aggregates the experiment results for each tested program and groups them concerning the manual (column "First Experiment"), the automatic mutation testing results with the reduction module disabled ("Second Experiment") and enabled ("Third Experiment"). For each program, the table shows the number of Spark transformations applied in the program (*Transf.*), the number of tests developed for the program (*Tests*), the number of generated mutants (*#M*), the number of killed mutants (*#K*) and the number of equivalent mutants (*#E*). Additionally, for the results obtained with the tool, the table shows the tool's execution time in seconds (*Time (s)*) and the number of mutants removed by the reduction module (*#R*). The evaluation and discussion of the reduction

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table V. Results of the experiments aggregated by program.

			First Experiment			Second Experiment				Third Experiment				
Program	**Transf.**	**Tests**	**#M**	**#K**	**#E**	**#M**	**#K**	**#E**	**Time (s)**	**#M**	**#K**	**#E**	**#R**	**Time (s)**
NGramsCount	5	5	27	22	5	32	27	5	243.2	23	22	1	9	176.8
ScanQuery	3	3	12	12	0	13	13	0	97.6	7	7	0	6	57.6
AggregationQuery	3	3	15	13	2	16	14	2	132.4	10	10	0	6	90.8
DistinctUserVisitsPerPage	4	2	16	10	6	17	11	6	142.6	11	7	4	6	98.2
MoviesRatingsAverage	5	4	25	22	3	19	14	5	182.8	13	11	2	6	135.6
MoviesRecomendation	12	5	37	33	4	56	41	15	524.4	35	24	11	21	355.2
JoinQuery	11	6	27	25	2	37	35	2	332.6	21	20	1	16	201.8
NasaApacheWebLogsAnalysis	7	4	55	49	6	38	28	10	323.6	31	21	10	7	286.2
Total	**50**	**32**	**214**	**186**	**28**	**228**	**183**	**45**	**1979.2**	**151**	**122**	**29**	**77**	**1402.2**

module will be discussed in the section devoted to answering the research question $RQ2$. The tool's total execution time includes the time it spends generating mutants, executing tests and generating reports.

Figure 13 shows the results of the three experiments aggregated by program. The figure compares the total number of mutants and equivalent mutants generated in the first (see columns *Mutants 1* and *Equivalent 1*) and second set of experiments (see columns *Mutants 2* and *Equivalent 2*). Figure 14 also presents the result of the three experiments, but aggregated by mutation operators.

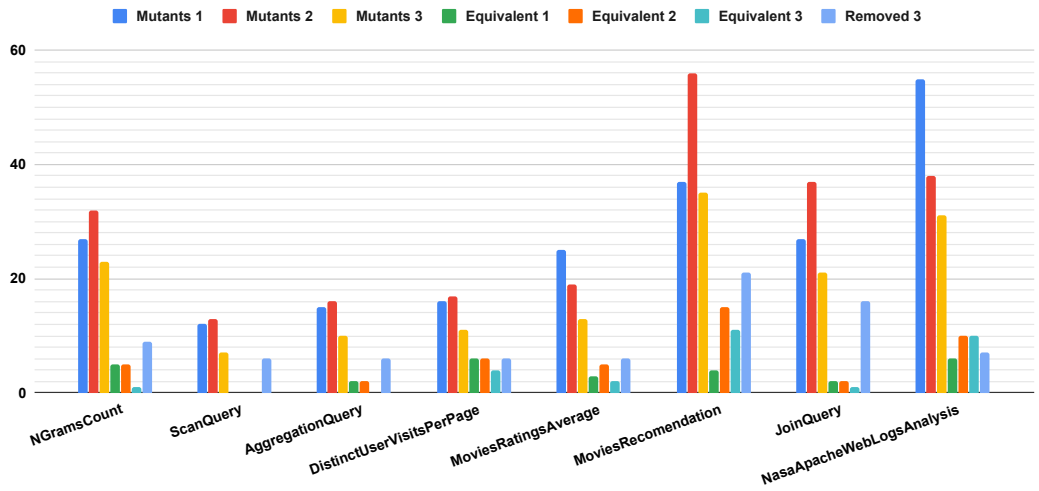


Figure 13. Comparison of aggregated results by program.

Analysis of the number of mutants One crucial remark before comparing the number of mutants of the first experiment and the ones with TRANSMUT-SPARK is the change in strategy concerning the scope of Spark transformations to be considered during mutation. Recall that TRANSMUT-SPARK treats subprograms as independent programs for generating mutants, thus reducing the

Prepared using stvrauth.cls
DOI: 10.1002/stvr

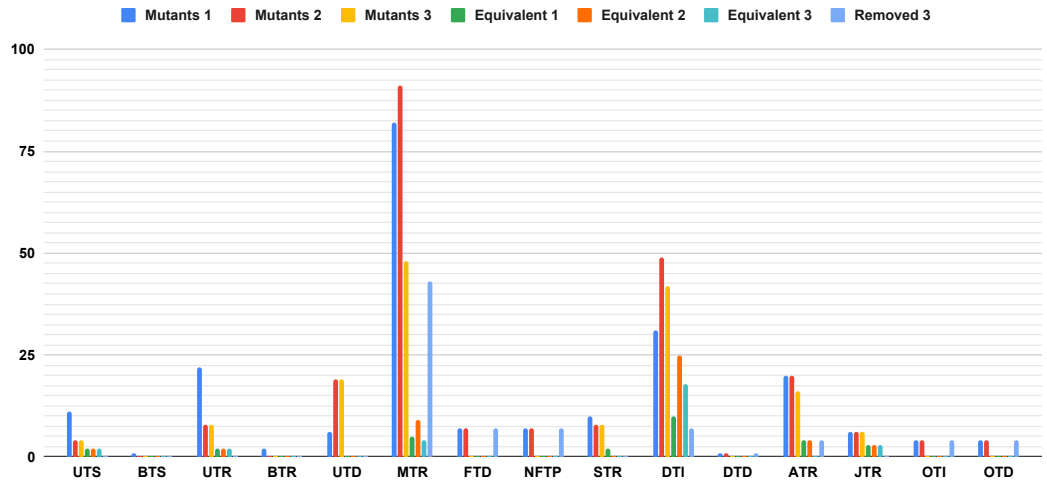


Figure 14. Comparison of aggregated results by mutation operator.

number of transformations available to generate new mutants. In theory, then, the number of mutants with TRANSMUT-SPARK should be smaller or equal to those of the first experiment if the tool were not more thorough in the application of mutants than humans. Indeed, the mutant generation process calls for a detailed analysis of target programs to identify possible modification points within the code the lines where mutation operators can be applied. This process is challenging given the number of generated mutants, making a manual generation prone to errors.

Figure 13 and Table V show that in most cases, the second experiment generated more mutants than the first experiment, with up to 51% increase, in the case of the program `MoviesRecomendation`. These results show that with TRANSMUT-SPARK we could indeed generate mutants that were not generated in the manual process and thereby answer RQ_1. Regarding the programs `NasApacheWebLogsAnalysis` and `MoviesRatingsAverage`, the second experiment generated around 30% and 24% fewer mutants than the first experiment. They were the programs where the change of strategy had the greater impact. Observation of the equivalent mutants shows that this number did not change in five programs of the original battery and was more significant in the second experiment for three of the programs.

When observing the differences concerning the mutants generated by the operator, we could see that some mutants were not generated in the manual process for UTD, MTR and DTI (see Figure 14

Prepared using stvrauth.cls
DOI: 10.1002/stvr

and Table VI). On the other hand, the data flow swap and replacement mutation operators (UTS, BTS, UTR, BTR), which consider other transformations on the same scope for mutation, generate fewer mutants. In the first experiment, the data flow mutation operators (UTS, BTS, UTR, BTR and UTD) generated approximately 20% of the mutants for the eight programs of the testing battery, against 13.6% on the second experiment. This reduction had no negative impact on the results of the second experiment because the tests that killed their mutants were also needed to kill mutants of other operators. Concerning the other mutation operators, the number of mutants generated was consistent in both experiments (with a slight difference for the STR operator where we automatically eliminated some equivalent mutants while implementing the tool).

A total of 195 mutants were generated, 24 of which were equivalent regarding the program `MovieLensExploration`, exclusively used by the TRANSMUT-SPARK experiments. This amount corresponds to approximately 85% of the total mutants generated for the other eight programs used in the experiments.

Analysis of the metric KR Table VI presents the results of the eight programs aggregated by the mutation operator. In it, we can see the number of generated mutants (*#M*), the number of equivalent mutants (*#E*) and the KR metric (*KR (%)*), as well as the number of mutants removed for the experiments with the reduction module enabled in the tool (*#R*).

Mutation operators did not significantly differ between the first and second experiments regarding KR. Exceptions observed are related to the change in strategy concerning the scope of Spark transformations to be considered during mutation: (1) the operators BTS and BTR did not generate mutants with TRANSMUT-SPARK; and (2) UTS generated fewer mutants and had a significant variation in its KR.

The operators with better results were UTS, DTI, DTD, JTR, OTI and OTD with low KR values (below 30%). These values show that fewer tests killed the mutants generated by them. In contrast, the mutants generated by operators with high KR values, like MTR and NFTP, had their mutants

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table VI. Results of the experiments aggregated by mutation operator.

	First Experiment			Second Experiment			Third Experiment			
Operator	**#M**	**#E**	**KR (%)**	**#M**	**#E**	**KR (%)**	**#M**	**#E**	**#R**	**KR (%)**
UTS	11	2	67.6	4	2	25.0	4	2	0	25.0
BTS	1	0	75.0	0	0	–	0	0	0	–
UTR	22	2	39.0	8	2	37.5	8	2	0	37.5
BTR	2	0	37.5	0	0	–	0	0	0	–
UTD	6	0	32.0	19	0	32.9	19	0	0	32.9
MTR	82	5	76.1	91	9	73.8	48	4	43	67.1
FTD	7	0	34.4	7	0	36.1	0	0	7	–
NFTP	7	0	65.6	7	0	66.6	0	0	7	–
STR	10	2	34.4	8	0	34.4	8	0	0	34.4
DTI	31	10	27.7	49	25	26.2	42	18	7	26.2
DTD	1	0	25.0	1	0	25.0	0	0	1	–
ATR	20	4	46.4	20	4	46.4	16	0	4	46.4
JTR	6	3	22.2	6	3	22.2	6	3	0	22.2
OTI	4	0	30.0	4	0	25.0	0	0	4	–
OTD	4	0	20.0	4	0	16.6	0	0	4	–

killed by a more significant amount of tests. This result intuitively indicates that these operators might be trivial.

The aggregated results by mutation operator of the program `MovieLensExploration` are shown in Table VII. In this program, a total of 33 Spark transformations are applied. We developed 17 test cases to kill the 171 non-equivalent mutants generated by TRANSMUT-SPARK for the program. The average execution time of the tool for this program was approximately 39.4 minutes (2364.6 seconds). The reduction module automatically removed 65 of the generated mutants. The tool's average execution time was approximately 24.6 minutes (1475.8 seconds) with this reduction.

As shown in Table VII, the KR values show that the mutants generated with the operators MTR, UTS and UTR were killed by most tests developed for the program `MovieLensExploration`. As in the first experiment, the mutants generated with the operator MTR were the ones that died more easily and contributed less in the test generation process since they did not need particular tests to kill them. In contrast, the operators DTI, JTR and OTD had the lowest KR, indicating that their mutants required more specific tests to be killed.

Analysis of the execution time of the mutation process The most significant difference between the first two experiments was the time spent executing the mutation testing process. The

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table VII. Results of the experiments for the program `MovieLensExploration` aggregated by mutation operator.

	Second Experiment			Third Experiment			
Operator	**#M**	**#E**	**KR (%)**	**#M**	**#E**	**#R**	**KR (%)**
UTS	3	0	88.8	3	0	0	88.8
UTR	6	0	94.4	6	0	0	94.4
UTD	8	0	69.2	8	0	0	69.2
MTR	114	10	99.2	66	5	48	98.6
DTI	32	11	43.4	25	6	7	42.8
ATR	20	3	73.9	16	0	4	72.7
JTR	6	0	22.2	6	0	0	22.2
OTI	3	0	58.3	0	0	3	–
OTD	3	0	41.6	0	0	3	–
Total	**195**	**24**	–	**130**	**11**	**65**	–

experiments' execution and analysis of the first experiment results took approximately four weeks, with an average of roughly three days for each program. This work involved the manual generation of the mutants, implementation of test cases, implementation of scripts for automatic execution of the mutants, and manual collection and analysis of the results, being most of these laborious and repetitive tasks.

The second experiment tested TRANSMUT-SPARK and showed it drastically reduces the execution time of the mutation testing process. The tool automates three tasks: generating the mutants, executing the tests, and analyzing the results. Thus, we put effort into developing test cases and analyzing living mutants to identify equivalent ones. The work that took on average three days for each program in the first experiment was done in a few hours in the second experiment. This effort was significant for the program `MovieLensExploration` that was not applied in the first experiment due to its complexity compared to the other programs, being impractical to use the process manually. Thus, with the aid of TRANSMUT-SPARK, the effort in the mutation testing process of `MovieLensExploration` was concentrated on the development of tests, which remains manual.

For the eight programs in the first experiment, the process with TRANSMUT-SPARK took a few minutes for each program (see Table V). We reused the tests of the first experiment; thus, we did not consider the time spent in defining them. The program `MoviesRecomendation` was the one that generated the most significant amount of mutants and took the longest time to finalize the process

Prepared using stvrauth.cls
DOI: 10.1002/stvr

(approximately nine minutes). Regarding the program `MovieLensExploration`, we spent approximately one day developing its tests, and the execution of the process took approximately 39 minutes on average.

Analysis of the generated mutants per operator We observed that the mutants generated by MTR were trivially killed, particularly in cases where mutants were mapped to Max, Min, "", $x.reverse$ and $null$. In contrast, MTR mutants mapped to other values had better results.

As discussed in Section 4, through the experiments, we confirmed the relation between couples of operators FTD/NFTP and OTD/OTI that shows that the tests that kill FTD/OTD mutants always kill NFTP/OTI mutants, but not the opposite. Finally, since UTD represents a general transformation removal, the mutants generated with the FTD, DTD and OTD operators were also generated by UTD. Thus, the application of the operator UTD overrides the application of FTD, DTD and OTD; otherwise, mutants are duplicated. These results were used to define the reduction rules implemented by the mutants reduction module. Table VII shows the number of mutants generated for each mutation operator applied to the program `MovieLensExploration`. Note that the operator MTR generated approximately 58% of the program's mutants. Of the 33 transformations applied in the program `MovieLensExploration`, 20 were mapping transformations (approximately 60%). The more significant mapping transformations explain the number of mutants generated by MTR. As in the first experiment, the operator DTI generated the second largest number of mutants and the most significant number of equivalent mutants.

7.2. *RQ2: Impact of Mutants Reduction in* TRANSMUT-SPARK *performance*

The strategy adopted to answer RQ_2 was to compare TRANSMUT-SPARK performance when reduction rules are applied for reducing the number of mutants.

Analysis of the impact of the mutants reduction module use Then columns *Mutants 3* and *Equivalent 3* in Figure 13 compare the number of mutants and equivalent mutants generated with TRANSMUT-SPARK using the reduction module. Finally, column *Removed 3* shows the number

Prepared using stvrauth.cls
DOI: 10.1002/stvr

of removed mutants by the reduction module. The differences between the results of the three experiments aggregated by mutation operator can be seen in Figure 14.

The analysis of the results of the first and second experiments motivated the development of the reduction rules introduced in Section 4 and the implementation of the reduction module of TRANSMUT-SPARK. This module applies a *selective mutation* strategy [53] to remove equivalent, redundant or trivial (i.e., mutants that are easily killed) mutants from the set of mutants generated by TRANSMUT-SPARK. This strategy reduces the number of mutants to be executed, which are more likely to contribute to the mutation testing process.

Table V and table VI show the results obtained by TRANSMUT-SPARK using the reduction module on the eight programs of the testing battery (*Third Experiment*). Recall that we used the same testing battery in the first and second experiments. Figure 13 and Figure 14 show the experimental result with the reduction model disabled and enabled. Note that the reduction module removed approximately 34% of the mutants generated by TRANSMUT-SPARK when the module was disabled. The module removed approximately 35% of equivalent mutants. The reduction of equivalent mutants reduces the effort required to analyze living mutants since equivalent mutants need to be detected manually.

The programs `MoviesRecomendation` and `JoinQuery` had approximately 37% and 43% fewer mutants than the second experiment, respectively. The mutants generated with the operators FTD, DTD, OTD, NFTP and OTI were removed using the reduction rules R_1, R_2 and R_3 (see Table IV). Then, approximately 47% of the mutants generated by the operator MTR were removed. For the operators DTI and ATR, all mutants removed by the module were equivalent.

The KR metric improved for the mutation operator MTR, reducing about 6.7%. The KR for the operators DTI and ATR did not change when equivalent mutants were removed, and the metric is not affected in this case. Finally, the removed mutants caused by the operators FTD, DTD, OTD, NFTP and OTI had no adverse effects on the process because their mutants were redundant with the mutants generated by the operator UTD. The application of these redundant mutants was unnecessary because the operator UTD was applied.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

For the program `MovieLensExploration`, the reduction module removed approximately 33% of the mutants generated by TRANSMUT-SPARK (see Table VII). In the case where only equivalent mutants are considered, this reduction was around 46%. The most significant reduction concerned the operator MTR, with 46% fewer mutants for the other eight programs. The KR of the operators MTR, DTI and ATR dropped approximately 1% compared to the KR obtained without the reduction module. Even if this variation was slight, the result is promising. It suggests that the module could remove inefficient mutants from the process without harmful side effects like not detecting failures.

One of the major impacts of the mutants reduction module was on the execution time. For the eight programs used in the first experiment, the module reduced the total execution time of the tool by 29%. For the program `MovieLensExploration`, this reduction was 37%. These tool execution time reductions were proportional to the number of mutants removed. In general, the third experiment's results show that the mutants reduction module improved TRANSMUT-SPARK results. The module could reduce the number of equivalent mutants and thereby the effort required for identifying them. The module could remove redundant and trivial mutants. The module improved the KR results for some mutation operators, which implies that it removed inefficient mutants. Finally, the module reduced the tool's execution time. Thus, the mutants reduction module achieved its goal by reducing the costs of the mutation testing process of TRANSMUT-SPARK.

7.3. RQ3: Comparison of TRANSMUT-SPARK *with Existing Scala Program Mutation Tools*

For RQ_3, we adopted the strategy of comparing experimentally TRANSMUT-SPARK with existing mutation tools and further assess the relevance of TRANSMUT-SPARK's underlying approach.

Comparison of* TRANSMUT-SPARK *with Scalamu Table VIII and Table IX show the performance of the test set designed to kill the mutants generated by TRANSMUT-SPARK ("TRANSMUT-SPARK Tests") and the performance of the test set designed to kill the mutants generated by *Scalamu* ("*Scalamu* Tests"). Besides, they show the number of tests designed for each program (*Tests*), the number of generated mutants (*#M*), the number of mutants killed (*#K*), the

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table VIII. Results obtained with TRANSMUT-SPARK using the reduction module for the comparative experiment of TRANSMUT-SPARK and *Scalamu*.

Program	TRANSMUT-SPARK Tests						*Scalamu* Tests					
	Tests	#M	#K	#E	#R	ms	Tests	#M	#K	#E	#R	ms
NGramsCount	5	23	22	1	9	1.00	4	23	18	1	9	0.82
ScanQuery	3	7	7	0	6	1.00	1	7	1	0	6	0.14
AggregationQuery	3	10	10	0	6	1.00	2	10	7	0	6	0.70
DistinctUserVisitsPerPage	2	11	7	4	6	1.00	1	11	7	4	6	1.00
MoviesRatingsAverage	4	13	11	2	6	1.00	6	13	7	2	6	0.64
MoviesRecomendation	5	35	24	11	21	1.00	9	35	24	11	21	1.00
JoinQuery	6	21	20	1	16	1.00	2	21	7	1	16	0.35
NasaApacheWebLogsAnalysis	4	31	21	10	7	1.00	3	31	15	10	7	0.71
MovieLensExploration	17	130	119	11	65	1.00	10	130	92	11	65	0.77
Total	**49**	**281**	**241**	**40**	**142**	**1.00**	**38**	**281**	**178**	**40**	**142**	**0.74**

Table IX. Results obtained with *Scalamu* for the comparative experiment of TRANSMUT-SPARK and *Scalamu*.

Program	TRANSMUT-SPARK Tests					*Scalamu* Tests				
	Tests	#M	#K	#E	ms	Tests	#M	#K	#E	ms
NGramsCount	5	15	11	1	0.79	4	15	14	1	1.00
ScanQuery	3	3	2	0	0.67	1	3	3	0	1.00
AggregationQuery	3	3	2	0	0.67	2	3	3	0	1.00
DistinctUserVisitsPerPage	2	1	1	0	1.00	1	1	1	0	1.00
MoviesRatingsAverage	4	28	20	2	0.77	6	28	26	2	1.00
MoviesRecomendation	5	44	34	1	0.79	9	44	43	1	1.00
JoinQuery	6	3	2	0	0.67	2	3	3	0	1.00
NasaApacheWebLogsAnalysis	4	7	5	0	0.71	3	7	7	0	1.00
MovieLensExploration	17	51	45	2	0.92	10	51	49	2	1.00
Total	**49**	**155**	**122**	**6**	**0.82**	**38**	**155**	**149**	**6**	**1.00**

number of equivalent mutants (*#E*), the number of mutants removed by the reduction module (*#R*) and the mutation score (*ms*). The mutation score values highlighted in blue indicate the cases with the same ms for the two test sets to insist on the comparison. The cases highlighted in green are those with an ms of 1.0. The cases highlighted in red are those with an ms below 1.0.

Analysis of the comparison of* TRANSMUT-SPARK *with Scalamu As shown in tables VIII and IX, the test set's performance was different for each tool. The columns "TRANSMUT-SPARK Tests" in Table VIII, and "Scalamu Tests" in Table IX show that both tools achieved a mutation score 1.0. This score value was expected, given that we developed a specific test set for each tool. In contrast, note that the mutation score (ms) of the test set designed for *Scalamu* achieved lower score values by TRANSMUT-SPARK and vice-versa. The test set designed to kill the mutants generated by *Scalamu* achieved an average ms of 0.74 killing TRANSMUT-SPARK mutants generated with

Prepared using stvrauth.cls
DOI: 10.1002/stvr

the reduction module enabled. The test set designed to kill the mutants generated by TRANSMUT-SPARK achieved an average ms of 0.82 killing *Scalamu* generated mutants. These results show that tests designed to kill TRANSMUT-SPARK mutants did not kill all of *Scalamu* mutants, and vice-versa. Thus, the results showed that one tool could complement the other since developing tests to kill the mutants of just one tool is not enough to kill all the mutants generated by the other.

7.4. Analysis and Discussion

The main goal of the experiments was assessing the use of a testing tool for testing Spark programs (compared with a less systematic and error-prone manual process). We also verified to which extent the tool allows more extensive, more realistic tests.

The manual experiment let us define a baseline that serves as a reference to assess two main aspects regarding the tool ([RQ_1 answer]):

1. Automation makes the mutation testing process more agile. It prevents programmers or people testing the tool to perform manual tasks far away from the objective of the task, which is assessing Spark data processing code. With the baseline as a reference, it is possible to determine to which extent an agile and automated process can: (i) lead to the generation of more mutants, (ii) calibrate the reduction strategy, (iii) easily compare the testing results, (iv) eventually activate/deactivate mutants, and (v) thoroughly analyse them understand and calibrate the testing process. Our results show that it is essential to have a tool with the facilities offered by TRANSMUT-SPARK within a testing process.
2. Keeping quantitative track of the testing process. TRANSMUT-SPARK also produces statistics that are useful to generate quantitative results about the testing process with metrics that can be representative to compare testing tasks under different conditions. A manual process would imply that programmers drag the pencil with a considerable burden to produce these testing results.

Concerning mutation operators through the tool's implementation, we could profile the set of operators, identifying those that override specific ones. This result let us define a "minimum" set of

Prepared using stvrauth.cls
DOI: 10.1002/stvr

mutation operators adapted to Spark programs. Identifying this "minimum" set is a significant result obtained through experiments that let us define the reduction module of TRANSMUT-SPARK. The goal of this module was achieved because the experiments showed that it contributed to reduce mutation testing process costs by removing inefficient mutants (*i.e.*, that did not contribute to the testing process) and reducing the tool's execution time ([RQ_2 answer]).

Traditional mutation testing and transformation mutation are complementary because they refer to different facets of the program and simulate different faults. In *traditional mutation testing*, modifications are made on the syntactic facet, and in consequence, it depends on the syntax of the programming language [16]. This approach mimics programming faults through syntactic deviations in the program, such as replacing one arithmetic operator with another. In contrast, transformation mutation simulates faults related to the definition of the data flow and specific transformations used in a data processing program independent of the programming language. The results of the TRANSMUT-SPARK and *Scalamu* comparison experiment confirm the hypothesis that both mutation approaches are complementary, and they are needed to thoroughly test a Scala program weaving code for processing data using Spark libraries operations ([RQ_3 answer]).

8. THREATS TO VALIDITY AND LIMITATIONS

The experimental validation presented in this work was the first attempt to evaluate our approach and tool. The experiments have some limitations and threats to their validity.

The first limitation is related to the testing battery. Programs implement code that focuses on the pure Big Data processing tasks within applications because TRANSMUT-SPARK tests this type of code and not the code that wraps these tasks. This choice implies that the programs of the testing battery are simple in terms of the number of lines, program logic, and scope, giving the impression of not being adapted for testing complex, industrial applications. Nevertheless, the complexity of the programs in the testing could increase by concatenating several Big Data processing tasks. However, this strategy does not seem realistic because, usually, applications do not combine several analytics

Prepared using stvrauth.cls
DOI: 10.1002/stvr

targets at a time. Besides, the phases of the analytics process like data preparation, cleaning, training and applying models [63] are treated as separated programs, just as we did with our testing battery.

Figure 5 shows that the mutation operators were defined based on the taxonomy resulting from a thorough study of Spark programs and Big Data processing operators. The use of the taxonomy prevents the empirical definition of mutation operators. In the same way, the experiments and faults used to define tests correspond to those of the taxonomy. This strategy ensures that experiments thoroughly test the families of faults that can come up in Spark Big Data processing code and assess the whole battery of operators proposed by our approach. The strategy is inspired from the mutation testing approach proposed by Ferrari *et al.* [64].

A second limitation for our study is related to the tests designed to kill mutants in experiments. Our goal in the experiments was to design tests that would kill all generated non-equivalent mutants. Thus, experiments provide a quantitative profile of the approach and its automation. The experiments are not defined to assess the approach and tool concerning other criteria (such as graph-based or input partitioning coverage [16]), and they do not provide insight to compare the results against other testing techniques. Also, the fact that the authors themselves develop the tests in the experiments may usually introduce a bias in the evaluation. We mitigated this threat by building the test set from simple tests as shown in Section 6.2 as a starting point and evolving the set with new simple tests when needed until reaching a mutation score of 1.0. We managed to avoid having unnecessary tests, and more importantly, unnecessarily complex tests. Consequently, the developed tests all contributed to the testing process and, at the same time, were not more powerful than required by the process.

Rationale of limitations and outreach TRANSMUT-SPARK relies on mutation operators, aimed at discovering specific faults (see Section 4). These operators only deal with (1) wrong data flow definition, and (2) inappropriate use of Spark built-in operations and their parameters. Aspects concerning misuse of variables for data sharing across distributed parallel processes deserves a thorough study and probably the design of an ad-hoc testing process and this is part of our

Prepared using stvrauth.cls
DOI: 10.1002/stvr

future work. TRANSMUT-SPARK also relies on reduction rules that apply a selective mutation approach [53] to remove redundant and inefficient mutants.

The current version of TRANSMUT-SPARK implements a simple strategy for managing test cases and mutants. Regarding tests, TRANSMUT-SPARK operates at the test class level, where a class can contain the implementation of one or more test cases. The tool allows specifying which test classes should be executed or not in the testing process. However, it cannot deal with a finer granularity, by selecting test cases within classes.

Regarding mutants, it is possible to define which operators will be applied or not in the process, but the tool does not allow to select and execute individual mutants. This is due to the fact that TRANSMUT-SPARK executes all the testing steps sequentially within a unique process (see Figure 6). The process is not interactive to let calibrate steps according to partial results—for example, select mutants to be executed between the generation and execution steps. Still, TRANSMUT-SPARK can execute a subset of mutants that survived a previous testing process. It also lets tag equivalent mutants to avoid unnecessary executions. TRANSMUT-SPARK allows tests to be added incrementally to the process, such that new tests can be developed to kill mutants that previous tests could not kill. These functionalities reduce the process execution time because only the necessary mutants are executed.

The program code TRANSMUT-SPARK can only test Spark programs built using the patterns it supports. Otherwise, programs have to be refactored before they are tested with TRANSMUT-SPARK. We decided to support specific patterns for the first version of TRANSMUT-SPARK to facilitate (1) *controllability* because the program can run independently of the context; (2) *observability* of the program elements (datasets and transformation) and behavior in the tests. These characteristics are fundamental to enable the automation of tests [16].

9. CONCLUSIONS AND FUTURE WORK

This paper introduced TRANSMUT-SPARK, a transformation mutation tool for Spark Big Data processing programs. TRANSMUT-SPARK automates the primary and most laborious steps of

Prepared using stvrauth.cls
DOI: 10.1002/stvr

the mutation testing process [12] and fully executes the testing process for Spark programs. TRANSMUT-SPARK implements operators for mutating the data flow and the transformations composing a Spark program [10]. TRANSMUT-SPARK deals with *test case handling* with the possibility for executing, including, and excluding test cases. TRANSMUT-SPARK also deals with *mutant handling* by generating, executing, and analyzing mutants. Finally, TRANSMUT-SPARK performs an adequacy analysis, calculating the mutation score and generating reports. The current version of TRANSMUT-SPARK is available on Github.

Through the description of the tool and experiments, the paper shows that TRANSMUT-SPARK is complementary to classical mutation testing tools addressing the data processing aspects of Spark programs. Experiments show that TRANSMUT-SPARK and *Scalamu* combined can lead to validation of both Scala code and weaved Spark data processing code. For the time being, the experiments run on TRANSMUT-SPARK have addressed representative data processing code, validating the mutation operators proposed in a previous work [10].

The assessment scores obtained experimentally (mutation score, killed ratio and process/execution time) showed promising results of TRANSMUT-SPARK in the process of testing Spark data processing programs. We can further use other testing batteries with programs that implement more processing tasks and compare results with other testing approaches. Our future work will tackle testing classic programs weaving data processing code using classical testing techniques (such as input space partitioning and logical coverage [16]) and with other work in the field [44].

Finally, our mutation operators [10] were formalized with the model for data flow programs presented in a previous paper [51], based on characteristics of different systems, including Apache Flink [2], Apache Beam [4], and DryadLINQ [3]. Thus, we also plan to extend TRANSMUT-SPARK to apply mutation testing to programs in other systems besides Spark.

ACKNOWLEDGEMENT

This study was partially funded by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

REFERENCES

1. Hadoop. Apache Hadoop Documentation 2019. URL https://hadoop.apache.org/docs/r2.7.3/.
2. Carbone P, Ewen S, Haridi S, Katsifodimos A, Markl V, Tzoumas K. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 2015; **38**(4):28–38.
3. Yu Y, Isard M, Fetterly D, Budiu M, Erlingsson U, Gunda PK, Currey J. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, USENIX Association: Berkeley, CA, USA, 2008; 1–14. URL http://dl.acm.org/citation.cfm?id=1855741.1855742.
4. Beam A. Apache Beam: An advanced unified programming model 2016. URL https://beam.apache.org/.
5. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: Cluster Computing with Working Sets. *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, USENIX Association: Berkeley, CA, USA, 2010; 10–10. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.
6. Garg N, Singla S, Jangra S. Challenges and Techniques for Testing of Big Data. *Procedia Computer Science* 2016; **85**:940 – 948, doi:https://doi.org/10.1016/j.procs.2016.05.285. URL http://www.sciencedirect.com/science/article/pii/S1877050916306354, international Conference on Computational Modelling and Security (CMS 2016).
7. Meeker WQ, Hong Y. Reliability Meets Big Data: Opportunities and Challenges. *Quality Engineering* 2014; **26**(1):102–116, doi:10.1080/08982112.2014.846119. URL https://doi.org/10.1080/08982112.2014.846119.
8. Liu J, Li J, Li W, Wu J. Rethinking big data: A review on the data quality and usage issues. *ISPRS Journal of Photogrammetry and Remote Sensing* 2016; **115**:134 – 142, doi:https://doi.org/10.1016/j.isprsjprs.2015.11.006. URL http://www.sciencedirect.com/science/article/pii/S0924271615002567, theme issue 'State-of-the-art in photogrammetry, remote sensing and spatial information science'.
9. DeMillo RA, Lipton RJ, Sayward FG. Hints on test data selection: Help for the practicing programmer. *Computer* April 1978; **11**(4):34–41, doi:10.1109/C-M.1978.218136.
10. Souza Neto JBd, Martins Moreira A, Vargas-Solar G, Musicante MA. Mutation Operators for Large Scale Data Processing Programs in Spark. *Advanced Information Systems Engineering*, Dustdar S, Yu E, Salinesi C, Rieu D, Pant V (eds.), Springer International Publishing: Cham, 2020; 482–497. URL https://doi.org/10.1007/978-3-030-49435-3_30.
11. Maldonado JC, Delamaro ME, Fabbri SCPF, da Silva Simão A, Sugeta T, Vincenzi AMR, Masiero PC. *Proteum: A Family of Tools to Support Specification and Program Testing Based on Mutation*. Springer US: Boston, MA, 2001; 113–116, doi:10.1007/978-1-4757-5939-6_19. URL https://doi.org/10.1007/978-1-4757-5939-6_19.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

12. Delamaro ME, Maldonado JC. Proteum-A Tool for the Assessment of Test Adequacy for C Programs. *Proceedings of the Conference on Performability in Computing Systems (PCS'96)*, New Brunswick, New Jersey, 1996; 79–95.
13. Spark. Apache Spark Documentation 2019. URL http://spark.apache.org/docs/2.2.0/.
14. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, USENIX Association: Berkeley, CA, USA, 2012; 2–2. URL http://dl.acm.org/citation.cfm?id=2228298.2228301.
15. Ganelin I, Orhian E, Sasaki K, York B. *Spark: Big Data Cluster Computing in Production*. Wiley, 2016.
16. Ammann P, Offutt J. *Introduction to Software Testing*. Second edition edn., Cambridge University Press: New York, NY, 2017.
17. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* Sep 2011; **37**(5):649–678, doi:10.1109/TSE.2010.62.
18. Frankl PG, Weiss SN, Hu C. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software* 1997; **38**(3):235 – 253, doi:https://doi.org/10.1016/S0164-1212(96)00154-9. URL http://www.sciencedirect.com/science/article/pii/S0164121296001549.
19. Offutt AJ, Pan J, Tewary K, Zhang T. An experimental evaluation of data flow and mutation testing. *Softw. Pract. Exper.* Feb 1996; **26**(2):165–176, doi:10.1002/(SICI)1097-024X(199602)26:2⟨165::AID-SPE5⟩3.0.CO;2-K. URL http://dx.doi.org/10.1002/(SICI)1097-024X(199602)26:2<165::AID-SPE5>3.0.CO;2-K.
20. Walsh PJ. A measure of test case completeness. PhD Thesis, State University of New York at Binghamton, Binghamton, NY, USA 1985. AAI8514636.
21. Teeuw W, Blanken H. Control versus data flow in parallel database machines. *IEEE transactions on parallel and distributed systems* Nov 1993; **4**(4):1265–1279, doi:10.1109/71.250104. Imported from EWI/DB PMS [db-utwente:arti:0000002027].
22. Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004; 137–150.
23. Camargo LC, Vergilio SR. Mapreduce program testing: a systematic mapping study. *Chilean Computer Science Society (SCCC), 32nd International Conference of the Computation*, 2013.
24. Morán J, de la Riva C, Tuya J. Testing MapReduce programs: A systematic mapping study. *Journal of Software: Evolution and Process* 2019; **31**(3):e2120, doi:10.1002/smr.2120. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2120.
25. Csallner C, Fegaras L, Li C. New ideas track: Testing mapreduce-style programs. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, ACM: New York, NY, USA, 2011; 504–507, doi:10.1145/2025113.2025204. URL http://doi.acm.org/10.1145/2025113.2025204.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

26. Li K, Reichenbach C, Smaragdakis Y, Diao Y, Csallner C. Sedge: Symbolic example data generation for dataflow programs. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013; 235–245.
27. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A. Pig latin: A not-so-foreign language for data processing. *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, ACM: New York, NY, USA, 2008; 1099–1110, doi:10.1145/1376616.1376726. URL http://doi.acm.org/10.1145/1376616.1376726.
28. Xu Z, Hirzel M, Rothermel G, Wu K. Testing properties of dataflow program operators. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013; 103–113.
29. Hirzel M, Andrade H, Gedik B, Jacques-Silva G, Khandekar R, Kumar V, Mendell M, Nasgaard H, Schneider S, Soulé R, *et al.*. Ibm streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development* 2013; **57**(3/4):7:1–7:11.
30. Morán J, d l Riva C, Tuya J. Mrtree: Functional testing based on mapreduce's execution behaviour. *2014 International Conference on Future Internet of Things and Cloud*, 2014; 379–384, doi:10.1109/FiCloud.2014.67.
31. Morán J, Riva Cdl, Tuya J. Testing Data Transformations in MapReduce Programs. *Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation*, A-TEST 2015, ACM: New York, NY, USA, 2015; 20–25, doi:10.1145/2804322.2804326. URL http://doi.acm.org/10.1145/2804322.2804326.
32. Mattos AJd. Test Data Generation for Testing MapReduce Systems. M.Sc. thesis, Universidade Federal do Paraná 2011.
33. Li N, Escalona A, Guo Y, Offutt J. A scalable big data test framework. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015; 1–2.
34. Chen YF, Hong CD, Sinha N, Wang BY. Commutativity of reducers. *Tools and Algorithms for the Construction and Analysis of Systems*, Baier C, Tinelli C (eds.), Springer Berlin Heidelberg: Berlin, Heidelberg, 2015; 131–146.
35. Chen YF, Hong CD, Lengál O, Mu SC, Sinha N, Wang BY. An executable sequential specification for spark aggregation. *Networked Systems*, El Abbadi A, Garbinato B (eds.), Springer International Publishing: Cham, 2017; 421–438.
36. DÖRRE J, APEL S, LENGAUER C. Static Type Checking of Hadoop MapReduce Programs. *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, Association for Computing Machinery: New York, NY, USA, 2011; 17–24, doi:10.1145/1996092.1996096. URL https://doi.org/10.1145/1996092.1996096.
37. Ono K, Hirai Y, Tanabe Y, Noda N, Hagiya M. Using Coq in Specification and Program Extraction of Hadoop MapReduce Applications. *Software Engineering and Formal Methods*, Barthe G, Pardo A, Schneider G (eds.), Springer Berlin Heidelberg: Berlin, Heidelberg, 2011; 350–365.
38. Bertot Y, Castran P. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. 1st edn., Springer Publishing Company, Incorporated, 2010.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

39. Brillout A, He N, Mazzucchi M, Kroening D, Purandare M, Rümmer P, Weissenbacher G. Mutation-based test case generation for simulink models. *International Symposium on Formal Methods for Components and Objects*, Springer, 2009; 208–227.
40. Movva V. Automatic test suite generation for scientific matlab code. M.Sc. thesis, University of Minesota 2015.
41. Xu Z, Hirzel M, Rothermel G, Wu KL. Testing properties of dataflow program operators. *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, IEEE Press, 2013; 103–113, doi:10.1109/ASE.2013.6693071. URL https://doi.org/10.1109/ASE.2013.6693071.
42. Karau H. Spark testing base 2015. URL https://github.com/holdenk/spark-testing-base.
43. Otto Group. Flinkspector 2016. URL https://github.com/ottogroup/flink-spector.
44. Riesco A, Rodríguez-Hortalá J. sscheck: Scalacheck for spark 2015. URL https://github.com/juanrh/sscheck.
45. Claessen K, Hughes J. Quickcheck: A lightweight tool for random testing of haskell programs. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, Association for Computing Machinery: New York, NY, USA, 2000; 268–279, doi:10.1145/351240.351266. URL https://doi.org/10.1145/351240.351266.
46. Riesco A, Rodríguez-Hortalá J. Temporal random testing for spark streaming. *Integrated Formal Methods*, Ábrahám E, Huisman M (eds.), Springer International Publishing: Cham, 2016; 393–408.
47. RIESCO A, RODRÍGUEZ-HORTALÁ J. Property-based testing for spark streaming. *Theory and Practice of Logic Programming* 2019; **19**(4):574–602, doi:10.1017/S1471068419000012.
48. Espinosa CV, Martin-Martin E, Riesco A, Rodríguez-Hortalá J. Flinkcheck: Property-based testing for apache flink. *IEEE Access* 2019; **7**:150 369–150 382.
49. Souza Neto JB. Transformation Mutation for Spark Programs Testing. PhD Thesis, Federal University of Rio Grande do Norte (UFRN), Natal-RN, Brazil 2020. (In Portuguese).
50. Suereth J, Farwell M. *SBT in Action: The Simple Scala Build Tool*. 1st edn., Manning Publications Co.: USA, 2015.
51. Souza Neto JB, Moreira AM, Vargas-Solar G, Musicante MA. Modeling Big Data Processing Programs. *Formal Methods: Foundations and Applications*, Carvalho G, Stolz V (eds.), Springer International Publishing: Cham, 2020; 101–118.
52. Usaola MP, Mateo PR. Mutation testing cost reduction techniques: A survey. *IEEE Software* 2010; **27**(3):80–86.
53. Offutt AJ, Rothermel G, Zapf C. An experimental evaluation of selective mutation. *Proceedings of 1993 15th International Conference on Software Engineering*, 1993; 100–107.
54. Untch RH, Offutt AJ, Harrold MJ. Mutation analysis using mutant schemata. *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '93, Association for Computing Machinery: New York, NY, USA, 1993; 139–148, doi:10.1145/154183.154265. URL https://doi.org/10.1145/154183.154265.
55. Offutt AJ, Untch RH. *Mutation 2000: Uniting the Orthogonal*. Springer US: Boston, MA, 2001; 34–44, doi:10.1007/978-1-4757-5939-6_7. URL https://doi.org/10.1007/978-1-4757-5939-6_7.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

56. Choi BJ, DeMillo RA, Krauser EW, Martin RJ, Mathur AP, Offutt AJ, Pan H, Spafford EH. The Mothra tool set (software testing). *[1989] Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume II: Software Track*, vol. 2, 1989; 275–284 vol.2, doi:10.1109/HICSS.1989.48002.
57. Odersky M, Spoon L, Venners B. *Programming in Scala: Updated for Scala 2.12*. 3rd edn., Artima Incorporation: Sunnyvale, CA, USA, 2016.
58. INFO SUPPORT. Stryker Mutator 2020. URL https://stryker-mutator.io.
59. Coles H, Laurent T, Henard C, Papadakis M, Ventresque A. Pit: A practical mutation testing tool for java (demo). *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, Association for Computing Machinery: New York, NY, USA, 2016; 449–452, doi:10.1145/2931037.2948707. URL https://doi.org/10.1145/2931037.2948707.
60. Sarwar B, Karypis G, Konstan J, Riedl J. Item-based Collaborative Filtering Recommendation Algorithms. *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, ACM: New York, NY, USA, 2001; 285–295, doi:10.1145/371920.372071.
61. AMPLab. Big data benchmark 2019. URL https://amplab.cs.berkeley.edu/benchmark/.
62. Harper FM, Konstan JA. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* Dec 2015; **5**(4):19:1–19:19, doi:10.1145/2827872.
63. Zöller MA, Huber MF. Benchmark and survey of automated machine learning frameworks. *Journal of Artificial Intelligence Research* 2021; .
64. Ferrari FC, Maldonado JC, Rashid A. Mutation testing for aspect-oriented programs. *2008 1st International Conference on Software Testing, Verification, and Validation*, 2008; 52–61, doi:10.1109/ICST.2008.37.

Prepared using stvrauth.cls
DOI: 10.1002/stvr