# Mutation Operators for Large Scale Data Processing Programs in Spark

João Batista de Souza Neto[1](✉), Anamaria Martins Moreira[2], Genoveva Vargas-Solar[3], and Martin Alejandro Musicante[1]

[1] Department of Informatics and Applied Mathematics (DIMAp), Federal University of Rio Grande do Norte, Natal, Brazil
jbsneto@ppgsc.ufrn.br, mam@dimap.ufrn.br

[2] Computer Science Department (DCC), Federal University of Rio de Janeiro, Rio de Janeiro, Brazil
anamaria@dcc.ufrj.br

[3] University Grenoble Alpes, CNRS, Grenoble INP, LIG-LAFMIA, Grenoble, France
genoveva.vargas@imag.fr

**Abstract.** This paper proposes a mutation testing approach for big data processing programs that follow a data flow model, such as those implemented on top of Apache Spark. Mutation testing is a fault-based technique that relies on fault simulation by modifying programs, to create faulty versions called *mutants*. Mutant creation is carried on by operators able to simulate specific and well identified faults. A testing process must be able to signal faults within mutants and thereby avoid having ill behaviours within a program. We propose a set of mutation operators designed for Spark programs characterized by a data flow and data processing operations. These operators model changes in the data flow and operations, to simulate faults that take into account Spark program characteristics. We performed manual experiments to evaluate the proposed mutation operators in terms of cost and effectiveness. Thereby, we show that mutation operators can contribute to the testing process, in the construction of reliable Spark programs.

**Keywords:** Big data · Spark programs · Mutation testing · Mutation operators

## 1 Introduction

The intrinsic characteristics of data and associated processing environments introduce challenges to the development of *big data* processing programs. These programs need to deal with data *Volume*; *Velocity* in which data is produced;

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

*Variety* of representation and *Veracity* level of the data. These technical characteristics, allied to the *Value* of the knowledge obtained by processing big data (the *five V's* [18]), have contributed to the development of systems and frameworks adapted to big data processing.

Existing frameworks adopt either control flow [4,6] or data flow approaches [3, 26,28]. In both cases, frameworks provide a complete and general execution environment that automates lower level tasks (processing and data distribution and fault tolerance), allowing developers to (mostly) concentrate on the algorithmic aspects of big data programs.

Reliability of big data processing programs becomes important, due to the fine-grain tuning required, regarding both the programming logic and particularly, their extensive use of computational resources [12]. This introduces the need to verify and validate programs before running them in production in a costly distributed environment. In this context, software testing techniques emerge as important and key tools. Testing big data processing programs is an open issue that is receiving increasing attention [5,20]. There exist only few works on functional testing of big data programs, most of them address testing of programs built using control flow based programming models like MapReduce [20].

This paper addresses big data programming testing by exploring the application of Mutation Testing on Apache Spark programs. Mutation testing is a fault-based technique that explores the creation of erroneous versions of a program, called *mutants*, to generate and evaluate tests. Mutants are created by applying modification rules, called mutation operators, that define how to create faulty versions from a program. In this paper, we present a set of mutation operators based on the data flow model of Apache Spark programs. We manually applied our mutation operators in an experiment to show the feasibility of mutation testing in Spark programs and to make a preliminary assessment of application costs and effectiveness of the proposed mutation operators. The results of these experiments agree with the preliminary results obtained using a prototype, currently under development[1].

This paper is organized as follows: Sect. 2 describes works that have addressed some aspects of big data program testing. Section 3 introduces the main concepts of mutation testing adopted in our work. Section 4 introduces Apache Spark and presents the set of mutation operators that we designed for Spark programs. Section 5 describes our experimental setting and discusses results. Section 6 concludes the paper and discusses future work.

[1] The description of the prototype is out of the scope of this paper. The interested reader can refer to https://github.com/jbsneto-ppgsc-ufrn/transmut-spark for technical details of the tool.

## 2 Related Work

The emerging need of processing big data together with the democratization of access to computing power has led to the proposal of environments providing solutions that ease the development of big data processing programs at scale. Even if these environments prevent programmers from dealing with the burden of low level control issues (e.g. fault tolerance, data and process distribution), programming must still consider several details regarding data flow (e.g., explicit data caching, exchange, sharing requests). Thus, testing methodologies must be proposed considering the particular characteristics of big data.

The testing of big data processing programs has gained interest as pointed out in [5] and [20]. Most work has focused on performance testing since performance is a major concern in a big data environment given the computational resources required [20]. Regarding functional testing, few works have been done, most of them being concentrated on MapReduce [20], leaving an open research area for testing big data programs on other models and technologies.

The work in [8] applies symbolic execution to search for test cases and data. It proposes to encode MapReduce correctness conditions into symbolic program constraints which are then used to derive the test cases and test data. The proposal in [16] applies a similar technique to test *Pig Latin* [22] programs. The MRFlow technique in [19] builds a data flow graph to define the paths to test and uses *graph-based testing* [1] to search for test cases in MapReduce.

Concerning data flow systems, most of the them support unit test execution for their programs. The work in [14] provides a framework that supports execution of unit testing and property checking of Spark programs. The tool does not provide support for the design of test cases, which is a critical part of the testing process. The area is still lacking techniques and tools that exploit the characteristics of big data processing programs, showing that more research needs to be done. Mutation testing provides criteria for the systematic design of test cases. In this context, our work explores both the design and application of mutation testing in Spark.

## 3 Mutation Testing

Mutation testing is a fault-based technique based on creating variants of a program, called *mutants*, simulating common faults inserted through simple modifications to the original program [1]. Mutants can be used to design test cases that identify the simulated faults or to assess the quality of an already implemented test set by looking if it can identify erroneous behaviors generated by the mutants. Different studies [11,21,25] have shown the effectiveness of mutation testing by comparing it with other testing criteria and techniques.

A general mutation testing process is shown in Fig. 1. Given a source program assumed to be nearly correct (developed by a competent programmer), Mutation Generation consists in creating variations of the program ( *Mutants*), by introducing changes or *mutations* to the source code. Examples of classic mutations include the replacement of literals in the program (*value mutations*); substitution of operators in expressions or conditions (*decision mutations*) and the deletion or duplication of statements (*statement mutations*) [1]. This phase of the testing process is strongly dependent on the model or language of the program being tested and on a number of *mutation operators*, rules which define how to derive mutants from the original program. The task of generating mutants of a given program can be automated using parsing and source-to-source code generation techniques.

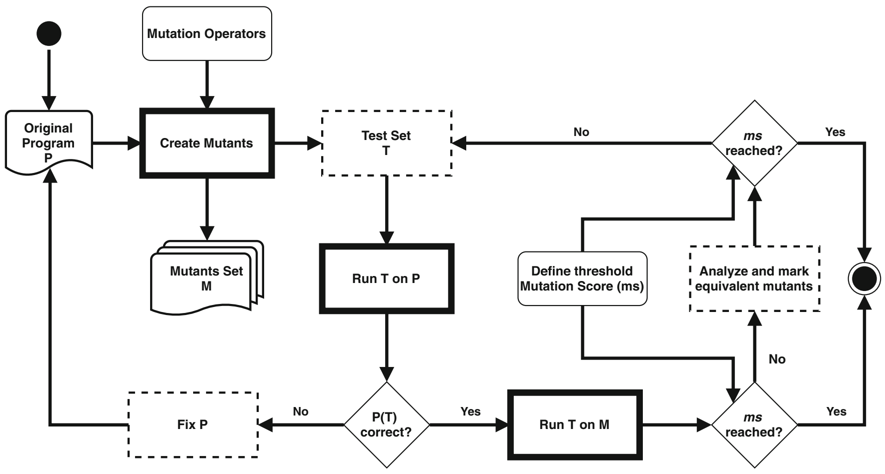

**Fig. 1.** Mutation testing process (Adapted from [1]).

The production of test cases is the next stage of the process. In this step input data, corresponding to each test case, is defined for the program and its mutants. The results of executing the test cases with each mutant are compared with the results obtained by the original program. A mutant is said to be *killed* if its results differ from those of the original program for some test case. The goal of a test set is then to kill as many mutants as possible. This indicates that the test set was able to detect the potential inserted code defects. Mutants that produce the same results as the original program, no matter which input data is provided, cannot be killed and are said to be *equivalent* to the original program.

Given a program $P$ and a set of test cases $T$, a *mutation score* is given by:

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

where $DM(P,T)$ is the number of killed mutants; $M(P)$ is the number of mutants and $EM(P)$ is the number of mutants that are *equivalent* to $P$. The mutation score measures the quality of the test set. This score is used to decide whether to produce more test cases, to improve the test set or to stop the testing process.

Mutation testing is strongly influenced by the programming model, language and framework of the target program. Thus, mutation operators and tools have been developed to support mutation testing for different contexts as shown in [13]. Such contexts include mutation operators and tools for programs in specific languages like C [23] and Java [17], aspect-oriented programs [10] and web services [15]. To the best of our knowledge, there is no previous work addressing mutation testing for data flow programs in the context of big data processing.

## 4 Testing Apache Spark Programs

Apache Spark is a general-purpose analytics engine for large-scale data processing on cluster systems [28]. It adopts a data flow-oriented programming model with data models, execution plans and programming interfaces with built in operations as building blocks for big data processing programs. Spark is centered on the concept of *Resilient Distributed Dataset* (RDD) [27], a read-only, fault-tolerant data collection that is partitioned across a cluster. RDDs can be processed by two kinds of operations: transformations and actions. Transformations are operations that result in a new RDD from processing another one. Actions are operations that generate values that are not RDDs or that save the RDD into an external storage system. Spark transformations are evaluated under a lazy strategy when an action is called.

A Spark program is defined as a set of initial RDDs loaded from an external storage, a sequence of transformations to be applied on these RDDs and actions that trigger the program execution. The sequence of operations implementing a Spark program is represented by a *Directed Acyclic Graph* (DAG) which acts as execution plan defining dependencies between transformations and representing the program data flow. These aspects are key elements for developing specific testing methodologies.

Spark provides a set of transformations for a wide variety of data processing operations. These transformations are described by a high-level interface with input parameters, which are functions that are applied to process elements on the RDD, and outputs. We classify transformations into families, according to the type of processing operation: *Mapping*, apply functions to map one element of the RDD to another (*e.g.*, map and flatMap); *Filtering*, filter elements based on predicate functions that determine whether an element should remain in the RDD (*e.g.*, filter); *Aggregation*, aggregate elements applying a binary function on the RDD elements (*e.g.*, reduceByKey and aggregateByKey); *Set*, operate like mathematical set operations on two RDDs (*e.g.*, union and intersection); *Join*, make a relational-like join between two RDDs (*e.g.*, (inner) join and leftOuterJoin); and *Ordering*, for operations that sort the elements on the RDD (*e.g.*, sortBy and sortByKey). We call unary transformations those that operate on a single RDD and binary transformations those that operate on two RDDs.

In order to propose a fault based approach for testing Apache Spark programs, we first studied representative example programs and the framework documentation to identify common faults or mistakes. Within Spark's programming approach, a program is defined by a *(i)* data flow that defines data transmission, sharing, caching and persistence strategies to be adopted by processes running on cluster components; and *(ii)* data processing operations. Considering this characteristic of Spark programs we have classified faults that can emerge within these complementary aspects and proposed a fault taxonomy. This taxonomy was then used as reference to the definition of the mutation operators that are a key element of our mutation testing based approach[2]. The mutation operators we propose have been designed considering the Spark data flow model and its operations (transformations). These components are independent of the programming language chosen to develop Spark programs (which can be done in Scala, Java or Python). Thus, our mutation operators are agnostic to the programming language and can be applied to any program that follows the data flow model of Spark. The next sections introduce these operators and their experimental validation.

## 4.1 Mutation Operators for Apache Spark Programs

Mutation operators are rules that define changes on a program to add simulated faults. These operators are designed to mimic common faults, such as a missing iteration of a loop or a mistake in an arithmetical or logical expression, or to prompt testers to follow common test heuristics, such as requiring a test where a specific action is executed [1].

Common faults and mistakes in Spark programs are generally related to the incorrect definition of the data flow of a program, such as calling transformations in a wrong order, and mistakes in specific transformations, such as calling the wrong transformation or passing the wrong parameter.

In this paper, we propose two groups of mutation operators: *data flow* and *transformations*. Data flow operators define modifications in the sequence of transformations of a program (*i.e.*, altering the flow of data between transformations). Transformation operators define modifications in specific groups of transformations, like replacing a transformation by another or changing a parameter. This section introduces these operators intuitively see their formal definition in [9].

**Mutation Operators for the Data Flow:** Operators in this class change the sequence of operations that defines the data flow of a Spark program.

*Unary Transformations Replacement (UTR)* - Replace one unary transformation for another with the same input and output signature (RDD types).

*Unary Transformation Swap (UTS)* - Swap the calls of two transformations of the program, provided that they have the same input and output signature.

[2] The fault taxonomy of Spark programs and the mutation operators formalization are out of the scope of this paper, but are described in [9].

*Unary Transformation Deletion (UTD)* - Bypass a transformation that has the same RDD type as input and output.

We also define operators similar to UTS and UTR, adapted to binary transformations: *Binary Transformation Swap* (BTS) and *Binary Transformations Replacement* (BTR).

To illustrate the mutation operators for data flow, let us consider the excerpt from a Spark program presented in Fig. 2. In this program we manipulate an integer RDD (input: RDD[Int]). Line 1 filters the even numbers of the dataset. Then, each number is mapped to its square (line 2). Finally, the RDD is sorted (line 3). All the mutants generated for this program by our data flow mutation operators are presented in Table 1. In that table, only the lines affected by the mutation are included. For instance, applying the UTS operator to the transformations in lines 1 and 2, results in the mutant 7 of Table 1. In this mutant, the filter transformation that was called on line 1 is swapped with the map transformation that was called on line 2 in the original program.

```
1  val even = input.filter(x => x % 2 == 0)
2  val square = even.map(x => x * x)
3  val sorted = square.sortBy(x => x)
```

**Fig. 2.** Example of part of a Spark program.

**Table 1.** Mutants generated with the data flow mutation operators.

| Id | Operator | Lines | Mutation |
|---|---|---|---|
| 1 | UTR | 1 | val even =input.map( x =>x * x ) |
| 2 | UTR | 1 | val even =input.sortBy( x =>x ) |
| 3 | UTR | 2 | val square =even.filter( x =>x % 2 ==0 ) |
| 4 | UTR | 2 | val square =even.sortBy( x =>x ) |
| 5 | UTR | 3 | val sorted =square.filter( x =>x % 2 ==0 ) |
| 6 | UTR | 3 | val sorted =square.map( x =>x * x ) |
| 7 | UTS | 1,2 | val even = input.map(x => x * x)<br>val square = even.filter(x => x % 2 == 0) |
| 8 | UTS | 1,3 | val even = input.sortBy(x => x)<br>val sorted = square.filter(x => x % 2 == 0) |
| 9 | UTS | 2,3 | val square = even.sortBy(x => x)%<br>val sorted = square.map(x => x * x) |
| 10 | UTD | 1 | val even =input |
| 11 | UTD | 2 | val square =even |
| 12 | UTD | 3 | val sorted =square |

**Mutation Operators for Transformations:** In this class of operators, changes are made in specific transformations in a Spark program. Table 2 provides examples of mutants that are generated with the operators presented below.

**Table 2.** Mutants generated with the transformation mutation operators.

| Id | Operator | Line | Mutation |
|---|---|---|---|
| 1 | MTR | 2 | val square =even.map(x =>0) |
| 2 | MTR | 2 | val square =even.map(x =>1) |
| 3 | MTR | 2 | val square =even.map(x =>Int.MaxValue) |
| 4 | MTR | 2 | val square =even.map(x =>Int.MinValue) |
| 5 | MTR | 2 | val square =even.map(x =>−(x * x)) |
| 6 | FTD | 1 | val even =input |
| 7 | NFTP | 1 | val even =input.filter(x =>!(x % 2 ==0)) |
| 8 | STR | – | val rdd3 =rdd1.intersection(rdd2) |
| 9 | STR | – | val rdd3 =rdd1.subtract(rdd2) |
| 10 | STR | – | val rdd3 =rdd1 |
| 11 | STR | – | val rdd3 =rdd2 |
| 12 | STR | – | val rdd3 =rdd2.union(rdd1) |
| 13 | DTD | – | val rdd4 =rdd3 |
| 14 | DTI | 1 | val even =input.filter(x =>x % 2 ==0).distinct() |
| 15 | DTI | 2 | val square =even.map(x =>x * x).distinct() |
| 16 | DTI | 3 | val sorted =square.sortBy(x =>x).distinct() |
| 17 | ATR | – | val rdd4 =rdd3.reduceByKey((x, y)=>x) |
| 18 | ATR | – | val rdd4 =rdd3.reduceByKey((x, y)=>y) |
| 19 | ATR | – | val rdd4 =rdd3.reduceByKey((x, y)=>x + x) |
| 20 | ATR | – | val rdd4 =rdd3.reduceByKey((x, y)=>y + y) |
| 21 | ATR | – | val rdd4 =rdd3.reduceByKey((x, y)=>y + x) |
| 22 | JTR | – | val rdd4 = rdd3.leftOuterJoin(rdd2) .map(x => (x._1, (x._2._1, x._2._2.getOrElse("")))) |
| 23 | JTR | – | val rdd4 = rdd3.rightOuterJoin(rdd2) .map(x => (x._1, (x._2._1.getOrElse(0), x._2._2))) |
| 24 | JTR | – | val rdd4 = rdd3.fullOuterJoin(rdd2) .map(x => (x._1, (x._2._1.getOrElse(0), x._2._2.getOrElse("")))) |
| 25 | OTD | 3 | val sorted =square |
| 26 | OTI | 3 | val sorted =square.sortBy(x =>x, ascending =false) |

*Mapping Transformation Replacement (MTR)* - for each mapping transformation (map, flatMap) in the program, replace the mapping function passed as a parameter to that transformation by a different mapping function. We propose a mapping function that returns a constant value of the same type as the original, or makes some modification to the value returned by the original function. For example, for a mapping function that operates on integers, we can replace this function by one that returns zero or another that reverses the sign of the value returned by the original. In Table 3 we present mapping values of basic types and collections that can be returned by the mutant mapping function. To illustrate the MTR operator, consider the mapping transformation applied in line 2 of Fig. 2. The operator generates mutants 1–5 in Table 2.

**Table 3.** 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 | $null$ |

Description: $x$ represents the value generated by the original mapping function; $k$ and $v$ represents the key and value generated by the original mapping function in case of Key-Value tuples; $k_m$ and $v_m$ represents modified values for the key and value, which is the application of other mapping values respecting the type.

*Filter Transformation Deletion (FTD)* - for each filter transformation in the program, create a mutant where the call to that transformation is deleted from the program. For example, considering the filter transformation in line 1 of Fig. 2, applying the FTD operator generates the mutation of line 6 in Table 2.

*Negation of Filter Transformation Predicate (NFTP)* - for each filter transformation in the program, replace the predicate function passed as a parameter to that transformation by a predicate function that negates the result of the original function. For the filter transformation in line 1 of Fig. 2, the NFTP operator generates the mutation 7 in Table 2.

*Set Transformation Replacement (STR)* - for each occurrence of a set transformation (union, intersection and subtract) in a program, create 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. For example, given the following excerpt of code with a union between two RDDs:

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

The application of the STR operator to this transformation creates the five mutants, described by lines 8–12 in Table 2.

*Distinct Transformation Deletion (DTD)* - for each call of a distinct transformation in the program, create a mutant by deleting it. 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 13 of Table 2:

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

*Distinct Transformation Insertion (DTI)* - for each transformation in the program, create a mutant inserting a distinct transformation call after that transformation. Applying DTI to the transformations presented in Fig. 2 generates the mutants 14–16 of Table 2.

*Aggregation Transformation Replacement (ATR)* - for each aggregation transformation in the program, replace the aggregation function passed as a parameter by a different aggregation function. We propose five replacement functions. For an original function $f(x, y)$, the replacement functions $f_m(x, y)$ are defined as: (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. For example, considering the following excerpt of code with an aggregation transformation (reduceByKey) and an aggregation function that adds two values, the application of ATR generates the mutants 17–21 of Table 2.

```
val rdd4 = rdd3.reduceByKey((x, y) => x + y)
```

*Join Transformation Replacement (JTR)* - for each occurrence of a join transformation ((inner) join, leftOuterJoin, rightOuterJoin and fullOuterJoin) in the program, replace that transformation by the remaining three join transformations. Additionally, a map transformation is inserted after the join to adjust the typing of the new join with the old one. This is necessary because depending on the join type, the left side, right side, or both can be optional, which makes the resulting RDD of the new join slightly different from the previous one. So we adjust the type of the resulting RDD to be of the same type as the original join. For example, replacing the join transformation by leftOuterJoin makes right-side values optional. To keep type consistency with the original transformation, we map empty right-side values to default values, in case of basic types, or null, otherwise.

To illustrate the JTR operator, let us consider the following code snippet where two RDDs are joined. Assume that rdd3 is of type RDD[(Int, Int)] and that rdd2 is of type RDD[(Int, String)]. The resulting RDD of this join (rdd4) is of type RDD[(Int, (Int, String))]. Applying JTR to this transformation generates the mutants 22–24 of Table 1. Taking mutant 22 as an example, replacing join with leftOuterJoin, the resulting RDD is of type RDD[(Int, (Int, Option[String]))]. Thus, the map following the leftOuterJoin serves to set the value of type Option[String] to String. When this value is empty (None), we assign the empty string ("").

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

*Order Transformation Deletion (OTD)* - for each order transformation (sortBy and sortByKey) in the program, create a mutant where the call to that transformation is deleted from the program. For example, considering the order transformation called in line 3 of Fig. 2, the application of OTD generates the mutant 25 of Table 2.

*Order Transformation Inversion (OTI)* - for each order transformation in the program, create a mutant where the ordering of that transformations is replaced by the inverse ordering (ascending or descending). Applying OTI to the same order transformation of Fig. 2 generates the mutant 26 of Table 2, where the ascending ordering that is true by default was changed for false.

## 5 Experiments

We conducted experiments to evaluate the cost and effectiveness of the proposed mutation operators. We selected a set of eight representative Spark programs[3] to apply the mutation testing process described in Fig. 1. These programs perform common data analysis such as text and log analysis, queries on tabular datasets inspired by the benchmark presented in [2], and data exploration and recommendation based on the collaborative filtering algorithm [24]. These programs were selected to explore the features necessary to apply the operators, such as having data flow and transformations commonly used in Spark programs and that could be modified in the testing process.

The experiments presented in this section show a first assessment of the mutation operators. The process described in Fig. 1 was strictly followed, by manually executing each step. For each code to be tested, and each applicable mutation operator, the source was edited to simulate the application of the operator, generating a mutant. A script was then executed to run each mutant. Test cases were developed incrementally to kill the mutants. Comparison of the results with the original program and metrics calculation were also executed manually. Once we implemented a prototype mutation testing tool, the results of these experiments evaluated and results were corroborated.

Finally, we performed a cost analysis based on the number of mutants generated and tests needed to kill all mutants ($ms = 100\%$). We also analyzed the effectiveness of the mutation operators by identifying the operators that generated mutants that were killed by most of the tests and operators that generated mutants that were harder to kill. Table 4 summarizes the results for each

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

program, showing the number of transformations in each program, number of mutants, number of tests created, number of killed mutants, number of equivalent mutants, and mutation score (*ms*).

**Table 4.** Total of mutants and tests per program.

| Program | Transformations | Mutants | Tests | Killed | Equiv. | *ms* (%) |
|---|---|---|---|---|---|---|
| NGramsCount | 5 | 27 | 5 | 20 | 5 | 100 |
| ScanQuery | 3 | 12 | 3 | 12 | 0 | 100 |
| AggregationQuery | 3 | 15 | 3 | 11 | 2 | 100 |
| DistinctUserVisitsPerPage | 4 | 16 | 2 | 10 | 6 | 100 |
| MoviesRatingsAverage | 5 | 25 | 4 | 22 | 3 | 100 |
| MoviesRecomendation | 12 | 37 | 5 | 33 | 4 | 100 |
| JoinQuery | 11 | 27 | 6 | 25 | 2 | 100 |
| NasaApacheWebLogsAnalysis | 7 | 55 | 4 | 49 | 6 | 100 |
| **Total** | **50** | **214** | **32** | **182** | **28** | – |

Table 5 summarizes the results aggregated for each mutation operator. It shows the total number of mutants generated by the operator, the number of equivalent mutants and the *killed ratio*[4]. The killed ratio shows how easy it was to kill the mutants generated with that mutation operator. Thus, operators with a low ratio generated mutants harder to kill (they required more specific tests). This measures the effectiveness of the mutation operator because mutants that are not killed trivially (get killed by any test) simulate faults that are not easily revealed.

**Table 5.** Total of mutants and killed ratio per mutation operator.

| **Mut. Op.** | **# of Mutants** | **# of Equiv.** | **Killed Ratio (%)** |
|---|---|---|---|
| UTS | 11 | 2 | 67,6 |
| BTS | 1 | 0 | 75,0 |
| UTR | 22 | 2 | 39,0 |
| BTR | 2 | 0 | 37,5 |
| UTD | 6 | 0 | 32,0 |
| MTR | 82 | 5 | 76,1 |
| FTD | 7 | 0 | 34,4 |
| NFTP | 7 | 0 | 65,6 |

| **Mut. Op.** | **# of Mutants** | **# of Equiv.** | **Killed Ratio (%)** |
|---|---|---|---|
| STR | 10 | 2 | 34,4 |
| DTI | 31 | 10 | 27,7 |
| DTD | 1 | 0 | 25,0 |
| ATR | 20 | 4 | 46,4 |
| JTR | 6 | 3 | 22,2 |
| OTI | 4 | 0 | 30,0 |
| OTD | 4 | 0 | 20,0 |

The mutation operators for data flow (UTS, BTS, UTR, BTR and UTD) were responsible for 19,6% of the generated mutants. The number of mutants

[4] The killed ratio is the ratio between the number of tests that killed the generated mutants and the total number of tests that were executed with those mutants.

generated by each of these operators depends on the number of transformations that have the same input dataset type and the same output dataset type. The number of mutants generated by UTS and BTS is equal to the number of two-by-two combinations between these transformations. In the case of the UTR and BTR, the number of mutants depends on the number of permutations of these transformations. The UTD generates a number of mutants equal to the number of transformations that have input and output datasets of the same type. From these operators, UTR, BTR and UTD generated the most relevant mutants since their mutants were killed by fewer tests.

The MTR operator generated the largest number of mutants (38,3% of total). Mapping operations are common in big data processing programs, which explains the number of mutants. The number of mutants depends on the type to which data is being mapped according to Table 3. For example, a mapping transformation that generates a numeric value will generate five mutants since we define five mapping values for numeric types. Analyzing the total, the mutants generated with the MTR were the easiest to kill, as we can see in Table 5. Individually, the mappings for 1, with numeric types, and $List(x.head)$ and $Nil$, in list type, obtained the best results with ratios below 70%.

The operators FTD and NFTP, related with filter transformations, and OTD and OTI, related with order transformations, generate a number of mutants equal to the number of transformations of the respective types. A subsumption relationship between FTD and NFTP, and between OTD and OTI was observed in the results. All tests that killed FTD mutants also killed NFTP mutants, just as all tests that killed OTD mutants also killed OTI mutants, but the opposite in both cases was not observed. This indicates that the FTD and OTD operators are stronger than the NFTP and OTI operators, which in turn indicates that when FTD and OTD are applied, the NFTP and OTI operators are not required.

The operator DTI generated 14.5% of all mutants, being the second operator that generated the most mutants. DTI is the most applicable operator because it can be applied after any transformation considering the resulting dataset is always the same type as the original. This operator also generated the largest number of equivalent mutants. This occurs because in some cases the operator is applied after aggregation transformations, so the repeated data had already been aggregated and the dataset no longer had duplicate data. In general, the DTI operator generated relevant mutants considering they were killed by less than 30% of the tests. The number of mutants generated with DTD is equal to the number of distinct transformations called in the program. In our experiment, only one program used distinct, which explains the existence of only one mutant.

ATR generated 11,2% of the mutants in the experiment. The number of mutants it generates is proportional to the number of aggregation transformations in the program, with five mutants for each transformation. The ATR operator has helped to improve the test set because it requires testing with diverse data to aggregate in order to kill the mutants. All equivalent mutants generated by ATR in our experiments were generated by the commutative replacement ($f_m(x, y) = f(y, x)$) because all aggregation functions applied in the programs were commutative. Even so, the commutative replacement mutation is useful because aggregation operations in distributed environments must be commutative to be deterministic [7]. Thus, this mutation can ensure that the commutative property is being taken into account in the testing process.

The mutation operators for binary transformations STR, for set-like transformations, and JTR, for join transformations, generate mutants proportional to the number of these types of transformations. A set transformation generate five mutants, whereas a join transformation generates three mutants. Both operators force the creation of test cases that include two input datasets with diversified data, containing data common to both as well as data not shared between them. In this manner, STR and JTR contribute to the improvement of the test set as they require nontrivial test data. This can be seen in Table 5 which shows that the killed ratio was 34,4% for STR and 22,2% for JTR, which we consider relevant results.

In general, the results showed a reasonable cost estimation for the mutation operators proposed in this work and the viability of their application in the mutation testing process. The number of mutants generated depends on the amount of transformations in the program and their types. The analysis of these aspects, as well as an analysis of the results shown in Table 5, such as the killed ratio, can be used as a reference to the selection of the operators to apply to the mutation testing of big data processing programs.

## 6 Conclusions and Future Work

The development of big data processing programs has gained interest in recent years. The distributed environment and computational resources required to run such programs make their costs high, which makes it necessary to validate and verify them before production. This paper addressed this issue by proposing the application of mutation testing to big data programs based on data flow systems.

We proposed a set of 15 mutation operators that take into account characteristics and operations of data flow systems to model changes in Spark programs and simulate faults. We applied these operators in an experiment to show their feasibility and make a first assessment of costs and effectiveness. The results showed the feasibility to apply mutation testing and design test cases for big data programs, at a reasonable cost. The experiment also hinted at the quality of mutation operators by showing which operators generated mutants, and hence faults, which were more difficult to identify, thus leading to more interesting test cases. This was revealed by the *killed ratio* for each operator in Table 5.

Our approach is complementary to traditional mutation testing criteria developed for Scala, Java and Python. The mutation analysis at the workflow level has several advantages. First, it reflects the two-level organization of Apache Spark programs, where the programmer defines the basic processing blocks (transformation) and the composition of these blocks (data flow). We designed our testing criteria to deal with this composition. Second, it can be used in addition to traditional mutation testing at the programming language level. Finally, it can be generalised to other data flow big data processing frameworks which have similarities in program definition and operations. Thus, we plan to do experiments to apply the proposed mutation operators for testing programs in other data flow frameworks such as Apache Flink [6], DryadLINQ [26] and Apache Beam [3].

Our work has shown that mutation testing can be successfully applied to big data processing programs by designing mutation operators specific to this class of programs. However, due to the effort required for generation, execution and analysis of the mutants, mutation testing is dependent on automation so that it can be viable. Thus, we are consolidating our prototype to automate the process of generation and execution of mutants, to assist the mutation testing of big data programs. Moreover, we plan to evaluate the effectiveness of our mutation operators by comparing the test sets created in the process with other test coverage criteria (e.g., input space partitioning and logic coverage [1]).

## References

1. Ammann, P., Offutt, J.: Introduction to Software Testing, 2nd edn. Cambridge University Press, New York (2017)
2. AMPLab: Big Data Benchmark (2019). https://amplab.cs.berkeley.edu/benchmark/
3. Apache Foundation: Apache Beam: An advanced unified programming model (2016). https://beam.apache.org/
4. Apache Foundation: Apache Hadoop Documentation (2019). https://hadoop.apache.org/docs/r2.7.3/
5. Camargo, L.C., Vergilio, S.R.: MapReduce program testing: a systematic mapping study. In: Chilean Computer Science Society (SCCC), 32nd International Conference of the Computation (2013)
6. Carbone, P., Ewen, S., Haridi, S., Katsifodimos, A., Markl, V., Tzoumas, K.: Apache Flink: stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015)
7. Chen, Y.-F., Hong, C.-D., Lengál, O., Mu, S.-C., Sinha, N., Wang, B.-Y.: An executable sequential specification for spark aggregation. In: El Abbadi, A., Garbinato, B. (eds.) NETYS 2017. LNCS, vol. 10299, pp. 421–438. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59647-1_31
8. Csallner, C., Fegaras, L., Li, C.: New ideas track: testing mapreduce-style programs. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 504–507. ACM (2011)
9. De Souza Neto, J.B.: An approach to mutation testing of big data processing programs. Thesis Proposal, Federal University of Rio Grande do Norte (2019). (in Portuguese). https://archive.org/details/prop-doc-joao-ufrn
10. Ferrari, F.C., Maldonado, J.C., Rashid, A.: Mutation testing for aspect-oriented programs. In: 2008 1st International Conference on Software Testing, Verification, and Validation, pp. 52–61, April 2008
11. Frankl, P.G., Weiss, S.N., Hu, C.: All-uses vs mutation testing: an experimental comparison of effectiveness. J. Syst. Softw. **38**(3), 235–253 (1997)
12. Garg, N., Singla, S., Jangra, S.: Challenges and techniques for testing of big data. Procedia Comput. Sci. **85**, 940–948 (2016). International Conference on Computational Modelling and Security (CMS 2016)

13. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
14. Karau, H.: Spark testing base (2019). https://github.com/holdenk/spark-testing-base
15. Lee, S.C., Offutt, J.: Generating test cases for XML-based Web component interactions using mutation analysis. In: Proceedings 12th International Symposium on Software Reliability Engineering, pp. 200–209, November 2001
16. Li, K., Reichenbach, C., Smaragdakis, Y., Diao, Y., Csallner, C.: SEDGE: symbolic example data generation for dataflow programs. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, pp. 235–245 (2013)
17. Ma, Y.S., Offutt, J., Kwon, Y.R.: MuJava: an automated class mutation system. Softw. Test. Verif. Reliab. **15**(2), 97–133 (2005)
18. Marr, B.: Big Data: Using SMART Big Data, Analytics and Metrics to Make Better Decisions and Improve Performance. Wiley, Hoboken (2015)
19. Morán, J., de la Riva, C., Tuya, J.: Testing data transformations in MapReduce programs. In: Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation, pp. 20–25. ACM (2015)
20. Morán, J., de la Riva, C., Tuya, J.: Testing MapReduce programs: a systematic mapping study. J. Softw. Evol. Process **31**(3), e2120 (2019)
21. Offutt, A.J., Pan, J., Tewary, K., Zhang, T.: An experimental evaluation of data flow and mutation testing. Softw. Pract. Exper. **26**(2), 165–176 (1996)
22. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: a not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 1099–1110. ACM (2008)
23. Richard, H.A., et al.: Design of mutant operators for the C programming language. Technical report (1989)
24. Sarwar, B., Karypis, G., Konstan, J., Riedl, J.: Item-based collaborative filtering recommendation algorithms. In: Proceedings of the 10th International Conference on World Wide Web, pp. 285–295. ACM, New York (2001)
25. Walsh, P.J.: A measure of test case completeness (software, engineering). Ph.D. thesis, State University of New York at Binghamton, Binghamton, NY, USA (1985)
26. Yu, Y., et al.: DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, pp. 1–14. USENIX Association (2008)
27. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, p. 2. USENIX Association (2012)
28. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud 2010, p. 10. USENIX Association (2010)