White-Box Testing of Big Data Analytics with Complex User-Defined Functions

Muhammad Ali Gulzar  
Shaghayegh Mardani  
University of California, Los Angeles  
USA

Madanlal Musuvathi  
Microsoft Research, USA

Miryung Kim  
University of California, Los Angeles  
USA

ABSTRACT

Data-intensive scalable computing (DISC) systems such as Google’s MapReduce, Apache Hadoop, and Apache Spark are being leveraged to process massive quantities of data in the cloud. Modern DISC applications pose new challenges in exhaustive, automatic testing because they consist of dataflow operators, and complex user-defined functions (UDF) are prevalent unlike SQL queries. We design a new white-box testing approach, called BigTest to reason about the internal semantics of UDFs in tandem with the equivalence classes created by each dataflow and relational operator.

Our evaluation shows that, despite ultra-large scale input data size, real world DISC applications are often significantly skewed and in-adequate in terms of test coverage, leaving 34% of Joint Dataflow and UDF (JDU) paths untested. BigTest shows the potential to minimize data size for local testing by 10^5 to 10^8 orders of magnitude while revealing 2X more manually-injected faults than the previous approach. Our experiment shows that only few of the data records (order of tens) are actually required to achieve the same JDU coverage as the entire production data. The reduction in test data also provides CPU time saving of 194X on average, demonstrating that interactive and fast local testing is feasible for big data analytics, obviating the need to test applications on huge production data.

CCS CONCEPTS

• Software and its engineering → Cloud computing: Software testing and debugging; • Information systems → MapReduce-based systems.

KEYWORDS

symbolic execution, dataflow programs, data intensive scalable computing, map reduce, test generation

1 INTRODUCTION

Data-intensive scalable computing (DISC) systems such as MapReduce [20], Apache Hadoop [1], Apache Spark [49] are commonly used today to process terabytes and petabytes of data. At this scale, rare and buggy corner cases frequently show up in production [50]. Thus, it is common for these applications to either crash after running for days or worse, silently produce corrupted output. Unfortunately, the common industry practice for testing these applications remains running them locally on randomly sampled inputs, which obviously does not flush out bugs hiding in corner cases.

This paper presents a systematic input generation tool, called BigTest, that embodies a new white-box testing technique for DISC applications. BigTest is motivated by the recent successes of systematic test generation tools [22, 24, 40]. However, the nature of DISC applications requires extending these in important ways to be effective. Unlike general-purpose programs addressed by existing testing tools, DISC applications use a combination of relational operators, such as join and group-by, and dataflow operators, such as map, flatmap, along with user-defined functions (UDFs) written in general purpose languages such as C/C++, Java, or Scala.

In order to comprehensively test DISC applications, BigTest reasons about the combined behavior of UDFs with relational and dataflow operations. A trivial way is to replace these dataflow operations with their implementations and symbolically execute the resulting program. However, existing tools are unlikely to scale to such large programs, because dataflow implementation consists of almost 700 KLOC in Apache Spark. Instead, BigTest includes a logical abstraction for dataflow and relational operators when symbolically executing UDFs in the DISC application. The set of combined path constraints are transformed into SMT queries and solved by leveraging an off-the-shelf theorem prover, Z3 or CVC4, to produce a set of concrete input records [11, 19]. By using such a combined approach, BigTest is more effective than prior DISC testing techniques [31, 34] that either do not reason about UDFs or treat them as uninterpreted functions.

To realize this approach, BigTest tackles three important challenges that our evaluation shows are crucial for the effectiveness of the tool. First, BigTest models terminating cases in addition to the usual non-terminating cases for each dataflow operator. For example, the output of a join of two tables only includes rows with keys that match both the input tables. To handle corner cases, BigTest carefully considers terminating cases where a key is only present in the left table, the right table, and neither. Doing so is crucial, as based on the actual semantics of the join operator, the output can contain rows with null entries, which are an important source of bugs. Second, BigTest models collections explicitly, which are created by flatmap and used by reduce. Prior approaches [31, 34] do...
not support such operators, and thus are unable to detect bugs if code accesses an arbitrary element in a collection of objects or if the aggregation result is used within the control predicate of the subsequent UDF. Third, BigTest analyzes string constraints because string manipulation is common in DISC applications and frequent errors are ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException during segmentation and parsing.

To evaluate BigTest, we use a benchmark set of 7 real-world Apache Spark applications selected from previous work such as PigMix[35], TITIAN [28], and BigSIFT [25]. While these programs are representative of DISC applications, they do not adequately represent failures that happen in this domain. To rectify this problem, we perform a survey of DISC application bugs reported in Stack Overflow and mailing lists and identify seven categories of bugs. We extend the existing benchmarks by manually introducing these categories of faults into a total of 31 faulty DISC applications. To the best of our knowledge, this is the first set of DISC application benchmarks with representative real-world faults. Such benchmarks are crucial for further research in this area.

We assess JDU (Joint Dataflow and UDF) path coverage, symbolic execution performance, and SMT query time. Our evaluation shows that real world datasets are often significantly skewed and inadequate in terms of test coverage of DISC applications, still leaving 34% of JDU paths untested. Compared to SEDGE [31], BigTest significantly enhances its capability to model DISC applications—in 5 out of 7 applications, SEDGE is unable to handle these applications at all, due to limited dataflow operator support and in the rest 2 applications, SEDGE covers only 23% of paths modeled by BigTest.

We show that JDU path coverage is directly related to improvement in fault detection capability—BigTest reveals 2X more manually injected faults than SEDGE on average. BigTest can minimize data size for local testing by 10^5 to 10^6 orders of magnitude, achieving the CPU time savings of 194X on average, compared to testing code on the entire production data. BigTest synthesizes concrete input records in 19 seconds on average for all remaining untested paths. Below, we highlight the summary of contributions.

- **BigTest** is the first piece of DISC white-box testing that comprehensively models dataflow operators and the internal paths of user-defined functions in tandem.
- **BigTest** makes three important enhancements to improve fault detection capability for DISC applications—(1) It considers both terminating and non-terminating cases of each dataflow operator; (2) It explicitly models collections created by flatmap and translates aggregation logic into an iterative aggregator; and (3) It models string constraints explicitly.
- **BigTest** puts forward a benchmark of manually injected DISC application faults along with generated test data, inspired by the characteristics of real world DISC application faults evidenced by Stack Overflow and mailing lists.
- **BigTest** finds 2X more faults than SEDGE, minimizes test data by orders of magnitude, and is fast and interactive.

Our results demonstrate that interactive local testing of big data analytics is feasible, and that developers should not need to test their program on the entire production data. For example, a user may monitor path coverage with respect to the equivalent classes of paths generated from BigTest and skip records if they belong to the already covered path, constructing a minimized sample of the production data for local development and testing.

The rest of the paper is organized as follows. Section 2 provides a brief introduction to Apache Spark and symbolic execution. Section 3 describes a motivating example. Section 4 describes the design of BigTest. Section 5 describes evaluation settings and results. Section 6 discusses related work. Section 7 concludes with future work.

### 2 BACKGROUND

**Apache Spark.** BigTest targets Apache Spark, a widely used data intensive scalable computing system. Spark extends the MapReduce programming model with direct support for dataflow and traditional relational algebra operators (e.g., group-by, join, and filter). Datasets can be loaded in Spark runtime using several APIs that create Resilient Distributed Datasets (RDDs), an abstraction of distributed collection [48]. RDDs can be transformed by invoking dataflow operations on them (e.g., `val filterRdd = rdd.filter(_ > 5)`). Dataflow operators such as `map`, `reduce`, and `flatMap` are implemented as higher-order functions that take a user defined function (UDF) as an input parameter. The actual evaluation of an RDD occurs when an action such as `count` or `collect` is called. Internally, Spark translates a series of RDD transformations into a Directed Acyclic Graph (DAG) where each vertex represents a transformation applied to the incoming RDD. The Spark scheduler executes each stage in a topological order.

**Symbolic Execution using Java Path Finder.** BigTest builds on Symbolic Java PathFinder (SPF) [37]. Internally, SPF relies on the

---

```scala
1 val x,y,z;
2 if(x<y){
3   z = y/x; //PC1: x < y = true, Effect: z=y/x
4   else
5   z = x/y; //PC2: x >= y = true, Effect: z=x/y
```

**Figure 1:** Symbolic PathFinder produces a set of path constraints and their corresponding effects

```scala
1 val trips = sc.textFile("trips_table.csv")
2 .map(x =>
3   val cols = s.split(",")
4     (cols(1),cols(3).toDouble/cols(4).toDouble) )
5 //Returns location and speed
6 val zip = sc.textFile("zipcode_table.csv")
7 .map(x =>
8     (cols(1),cols(2))
9   // Returns location and its name
10   .filter{
11     x => x._2 == "Palm"
12   } join
13   joined = trips.join(zip)
14   .map(x =>
15     if (x._2._1 > 40) ("tax",1)
16     else if (x._2._1 > 15) ("bus",1)
17     else ("walk",1)
18   })
19 .reduceByKey(_+_
20 .saveAsTextFile("hdfs://...")
```

**Figure 2:** Alice’s program estimates the total number of trips originated from “Palm”. 
analysis engine of Java PathFinder (JPF) model checking [44]. It interprets Java bytecode on symbolic inputs and produces a set of symbolic constraints. Each constraint represents a unique path in the program, and can be ingested by a theorem solver to generate test inputs. Figure 1 illustrates an example symbolic execution result. By attaching listeners to SPF, the path conditions and the effects of each path can be captured. For this program, SPF produces two path conditions: (1) the first path produces the effect of \( z = y/x \), when the path condition \( x < y \) holds true and (2) the second path produces \( z = y/x \) as an effect, when the path condition \( x \geq y \) is satisfied.

3 MOTIVATING EXAMPLE

This section presents a running example to motivate BigTest. Suppose that Alice writes a DISC application in Spark to analyze the Los Angeles commuting dataset. She wants to find the total number of trips originating from the “Palms” neighborhood using: (1) a public transport whose speed is assumed to be faster than 15 but slower than 40 mph, (2) a personal vehicle which is estimated to be faster than 40 mph, and (3) on foot which is estimated as slower than 15 mph. Each row in the Trips dataset represents a unique identifier for a trip, the start and end location in terms of a zip code, the trip distance in miles, and the trip duration in hours, for example, \([1, 90034, 90024, 10, 15] \) To map an area zip code to its corresponding area name, Alice uses another dataset that assigns a name to each zip code in the following manner: \([90034, Culver City] \)

To perform this analysis, Alice writes a Spark application in Figure 2. She loads both datasets (lines 1 and 6), parses each dataset, selects the start location of a trip as a key, and computes the average speed as a value by dividing the distance by duration (lines 2-4). Alice outputs a zip code as a key and an area name as a value (lines 7-9) and filters the area name with “Palms” at line 12. She joins the two data sets (line 13). In the subsequent map operation (line 15-18), she categorizes the trips based on the average speed into three categories. She finally counts the frequency of each trip kind and stores them (lines 20 and 21). Though this program is only 21 lines long, it poses several challenges for modeling test paths.

### Equivalence Classes of Dataflow Operators

Consider filter \( \Phi \) at line 11. To exhaustively test this operator, we must consider two equivalence classes: the first where a data record satisfies the filter and moves onto the next operator and the second where the filter does not satisfy and its data flow terminates. If we only model non-terminating case then test data would contain passing data records only and hence, would not detect a fault in which filter is removed from the DISC application. To model join at line 13, we must have three equivalence classes—two terminating cases and one non-terminating case: (1) an input record in the left table (“Trip”) does not have a matching key on the right table (“ZipCode”), terminating its data flow, (2) an input in the right table does not have a matching key on the left, terminating its data flow, and (3) there exists a key that appears in both tables, passing the joined result to the next operator. Modeling such terminating cases is crucial otherwise test data generated produce the same output for both join and leftOuterJoin and do not reveal faults that are based on incorrect join type usage.

### UDF Paths

Consider the last map \( \Phi \) at lines 15-18. There are three internal path conditions: (1) the speed > 40 mph, (2) the speed is > 15 mph and \( \leq 40 \) mph, and (3) the speed is \( \leq 15 \) mph. The sub figure \( \Phi \) in Figure 3b shows corresponding path conditions and effects.

### String Constraints

To analyze the second map \( \Phi \) at lines 7 to 9, we must reason about the entailed string constraints. Given a string \( z \) in sub figure \( \Phi \) in Figure 3b and 3a, to split the data into two columns, it must satisfy a string constraint \( z \text{split}("","",") \geq 2 \) to produce the effect where the key \( k_2 \) is \( z \text{split}("","",") (1) \) and the value \( v_2 \) is \( z \text{split}("",") (0)) \). String manipulation is critical to many DISC applications. In the above example, at least one test must contain a string \( z \) without delimiter ",", so that \( z \text{split}("",") (1) \) leads to ArrayIndexOutOfBoundsException which will then expose the inability of the UDF to handle exceptions.
While these example data records may not look realistic, such data where a node calls each UDF and combines the symbolic execution of a DISC application along with the internal symbolic execution results of the DISC application. Therefore, synthetic data is necessary and crucial to expose downstream program behavior.

4 APPROACH

BigTest takes an Apache Spark application in Scala as an input and generates test inputs to cover all paths of the program up to a given bound by leveraging theorem provers Z3 [19] and CVC4 [11].

4.1 Dataflow Program Decomposition

A DISC application is comprised of a direct acyclic graph where each node represents a dataflow operator such as reduce and corresponding UDFs. As the implementation of dataflow operators in Apache Spark spans several hundred thousand lines of code, it is not feasible to perform symbolic execution of a DISC application along with the Spark framework code. Instead, we abstract the internal implementation of a dataflow operator in terms of logical specifications. We decompose a DISC application into a dataflow graph where a node calls each UDF and combine the symbolic execution of the UDFs using the logical specification of dataflow operators.

UDF Extraction. BigTest compiles the DISC application into Java bytecode and traverses each Abstract Syntax Tree (AST) to search for a method invocation corresponding to each dataflow operator. The input parameters of such method invocation are UDFs represented as anonymous functions as illustrated in Figure 4b. BigTest

<table>
<thead>
<tr>
<th>#</th>
<th>CONSTRAINT</th>
<th>TRIPS</th>
<th>ZIPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>T.split(*).length &gt;= 5</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C2</td>
<td>T.split(<em>).length &gt;= 5 ∧ NotInt(T.split(</em>)(3))</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C3</td>
<td>T.split(<em>).length &gt;= 5 ∧ isInt(T.split(</em>)(4)) ∧ T.split(*)(4).toInt = 0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C4</td>
<td>T.split(*).length &gt;= 2 ∧ V1 == &quot;Palms&quot;</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C5</td>
<td>T.split(*).length &gt;= 2 ∧ V2 == &quot;Palms&quot;</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C6</td>
<td>T.split(*).length &gt;= 2 ∧ V2 == &quot;Palms&quot;</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C7</td>
<td>A.isInt(T.split(<em>)(4)) ∧ T.split(</em>)(4).toInt = 0</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>C8</td>
<td>A.isInt(T.split(<em>)(4)) ∧ T.split(</em>)(4).toInt != 0</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Table 1: Generated input data where each row represents a unique path. Variables T, V, and K are defined in Figure 3a.

1. sc.textFile("zipcode.csv").map(...)
2. .filter(_._2 == "Palms")

(a) DISC Application

(b) Generated AST

(c) Extracted Filter UDF

Figure 4: BigTest extracts UDFs corresponding to dataflow operators through AST traversal.

stores the UDF as a separate Java class shown in Figure 4c and generates a configuration file required by JPF for symbolic execution. BigTest also performs dependency analysis to include external classes and methods referenced in the UDF.

4.2 Logical Specifications of Dataflow Operators

This section describes the equivalence classes generated by each dataflow operator’s semantics. We use C1 to represent a set of path constraints on the input data, I, for a particular operator. A single element ci in C1 contains path constraints that must be satisfied to

Example: Consider the following Scala code:

```scala
def f(a:Int, b:Int){
  return a+b;
}
```

(a) Normal invocation of `f` with `reduce` with a corresponding UDF.

```
...reduce(f)
```

(b) Equivalent iterative version with a bound K

Handling Aggregator Logic. For aggregation operators, the attached UDF must be transformed. For example, the UDF for sum is associative binary function, which performs incremental aggregation over a collection shown in Figure 5a. We translate it into an iterative version with a loop shown in Figure 5b. To bound the search space of constraints, we bound the number of iterations to a user provided bound K (default is 2).

```
...reduce(arr)
```

Figure 5: (a) A normal invocation of `reduce` with a corresponding UDF. (b) An equivalent iterative version with a bound K

After BigTest extracts UDFs corresponding to dataflow operators through AST traversal,
Table 2: $C_I$ represents a set of incoming constraints from the input table $I$, whereas each constraint $c \in C_I$ represents a non-terminating path. $t(c)$ represents that record $t \in I$ must satisfy constraint $c$. $f$ defines the set of path constraints generated by symbolically executing $udf$ and $f(t)$ represents the path constraint of a unique path exercised by input tuple $t$.

![Figure 7: An iterative version of aggregator UDF](image-url)

![Figure 6: A UDF with string manipulation](image-url)
A naïve SPF does not handle collections well and thus may generate a cycle in the control flow graph, we finitize the loop node from (a program point and an edge for a vertex) constructed with a set of edges connecting dataflow operators. Imagine a DISC application in the incoming path conditions of an upstream operator.

For example, an expression involves a division operator, division by zero (i.e., \( x \neq 0 \)).

Exceptions. BigTest extends SPF to explicitly model exceptions. For example, when an expression involves a division operator, division by zero (i.e., \( x \neq 0 \)) is not possible to detect the fault using an array of length 1.

4.4 Test Data Generation

BigTest rewrites path constraints into an SMT query. For constraints on integer variables, BigTest uses analogous arithmetic and logical operators available in SMT. For string constraints, BigTest uses operations such as \( \text{str}++.\text{str.to.int} \), \( \text{str.at} \) and \( \text{splitn} \). BigTest introduces a new symbolic operation \( \text{splitn} \). If a path constraint contains a clause \( v = \text{splitn} \), BigTest generates (assert \( (s = \text{splitn} \) ) ) that is equivalent to \( v = \) \( v \). The string path conditions produced by BigTest do not contain arrays and instead model individual elements of an array up to a given bound \( K \).

BigTest generates interpreted functions for Java native methods not supported by Z3. For example, BigTest replaces \( \text{isInteger} \) with an analogous Z3 function. BigTest executes each SMT query separately and finds satisfying assignments (i.e., test inputs) to exercise a particular path. While executing each SMT query independently may lead to redundant solving of overlapping constraints, in our experiments, we do not find it as a performance bottleneck. Theoretically, the number of path constraints increases exponentially due to branches and loops; however, empirically, our approach scales well to DISC applications, because UDFs tend to be much smaller (in order of hundred lines) than DISC frameworks and we abstract the framework implementation using logical specifications.

Figure 8 shows an SMT query produced by BigTest for Figure 2. Lines 1 to 6 construct the first table to have four segments and the second table to have two segments separated by a comma. Lines 7 to 10 restrict a string to be a valid integer. To enforce such constraint that crosses the boundary of strings and integers, BigTest uses a custom function \( \text{isInteger} \) and Z3 function \( \text{str.to.int} \).
Table 3: Subject Programs

<table>
<thead>
<tr>
<th>#</th>
<th>SUBJECT PROGRAM</th>
<th>OUTPUT</th>
<th># OF OPERATORS</th>
<th>OPERATORS</th>
<th>PROGRAM CHARACTERISTICS</th>
<th>STRING PASSING</th>
<th># BRANCHES</th>
<th># UDFs</th>
<th>JDU PATHS (kx2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>IncomeAggregate</td>
<td>total income of individuals earning ≤$300 weekly</td>
<td>3</td>
<td>map, filter, reduce</td>
<td>✓</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>MovieRatings</td>
<td>total number of movies with rating ≥ 4</td>
<td>4</td>
<td>map, filter, reduceByKey</td>
<td>✓</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>AirportLayover</td>
<td>total layover time of passengers per airport</td>
<td>3</td>
<td>map, filter, reduceByKey</td>
<td>✓</td>
<td>2</td>
<td>4</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>CommuteType</td>
<td>total number of people using each form of transport for daily commute</td>
<td>6</td>
<td>map, filter, join, reduceByKey</td>
<td>✓</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>PigMix-L2</td>
<td>PigMix performance benchmark</td>
<td>5</td>
<td>map, join</td>
<td>✓</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>Grade Analysis</td>
<td>List of classes with more than 5 failing students</td>
<td>5</td>
<td>flatmap, filter, reduceByKey, map</td>
<td>✓</td>
<td>2</td>
<td>3</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>P7</td>
<td>WordCount</td>
<td>finds the frequency of words</td>
<td>3</td>
<td>flatmap, map, reduceByKey</td>
<td>✓</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9: JDU path coverage of BigTest, Sedge, and the original input dataset

Lines 11 to 14 enforce a record to contain “Palms” and the speed to be less than or equal to 15. Lines 15 to 19 join these constraints generated from a UDF to the subsequent dataflow operator.

5 EVALUATION

We evaluate the effectiveness and efficiency of BigTest using a diverse set of benchmark DISC applications. We compare BigTest against Sedge in terms of path coverage, fault detection capability, and testing time. We compare test adequacy, input data size, and potential time saving against three alternative testing methods: (1) random sampling of \( k \% \) records, and (2) using a subset of the first \( k \% \) records, and (3) testing on the entire original data.

- To what extent BigTest is applicable to DISC applications?
- How much test coverage improvement can BigTest achieve?
- How many faults can BigTest detect?
- How much test data reduction does BigTest provide?
- How long does BigTest take to generate test data?

Subject Programs. In terms of benchmark programs, we use seven subject programs from earlier works on testing [31] and debugging DISC applications [25, 28], listed in Table 3. The PigMix benchmark package contains a data generator script that generates large scale datasets. We utilize map and flatmap with UDFs in Apache Spark to translate unsupported Pig operators like loadAs and split.

Three programs MovieRating (P2), AirportLayover (P3), and WordCount (P7) are adapted from BigSIFT [25]. Each program is paired with a large scale dataset. The rest are self-created custom Apache Spark applications to add heterogeneity in dataflow operators and UDFs. Table 3 shows detailed descriptions of subject programs. All applications (1) involve complex string operations including split, substring, and toInt, (2) perform complex arithmetics, (3) use type Tuple for key-value pairs, and (4) generate and process a collection with custom logic using flatmap.

Experimental Environment. We run all large-scale data processing on a 16-node cluster. Each node is running at 3.40GHz and equipped with 4 cores, 32GB of RAM, and 1TB of disk capacity allowing us to run up to 120 tasks simultaneously. For storage, we use HDFS version 1.0.4 with a replication factor of 3. Due to a very small size

Figure 10: JDU path coverage of BigTest in comparison to alternative sampling methods of test data generated by BigTest, we leverage Apache Spark’s local running mode to perform experiments on a single machine.

5.1 Dataflow Program Support

BigTest supports a variety of dataflow operators prevalent in DISC applications. For instance, Apache Spark provides flatmap and reduceByKey for constructing and processing collections. The previous approach Sedge is designed for Pig Latin with only a limited set of operators support [31]. Sedge is neither open-source nor have any implementation available for Apache Spark for direct comparison. Therefore, we faithfully implement Sedge precisely based on the technical details provided elsewhere [31]. We manually downgrade BigTest by removing symbolic execution for UDFs and equivalence classes for certain operators to emulate Sedge. The implementation of both Sedge and BigTest are publicly available. Out of seven benchmark applications written in Apache Spark, five applications contain flatmap and reduceByKey, therefore, Sedge is not able to generate testing data for these 5 applications.

5.2 Joint Dataflow and UDF Path Coverage

We evaluate code coverage of BigTest, Sedge, and the original input dataset based on JDU path coverage defined in Section 4.3.

JDU Path Coverage Evaluation. We compare BigTest with three alternative sampling techniques: (1) random sampling of \( k \% \) of the original dataset, (2) selection of the first \( k \% \) of the original dataset, as developers often test DISC applications using head\(-n\), and (3) a prior approach Sedge. To keep consistency in our experiment setting, we enumerate JDU paths for a given user-provided bound \( K \) and measure how many of these paths are covered by each approach.

Figure 9 compares the test coverage from BigTest, Sedge, and the original input dataset. Y axis represents the normalized JDU path coverage ranging from 0% to 100%. Across seven subject programs, we observe that Sedge covers significantly fewer JDU paths (22% of what is covered by BigTest). By not modelling the internal paths of UDFs, Sedge fails to explore many JDU paths. Even when

1https://github.com/maligulzar/BigTest
Figure 11: The number of JDU paths covered and the test execution time when $k$% of the data is randomly selected and the first $k$% of data is selected for subject program ComuteType, the complete dataset is used, the JDU path coverage reaches only 66% of what BigTest could achieve. The entire dataset achieves better coverage than Sedge but it still lacks coverage compared to BigTest. In other words, using the entire big data for testing does not necessarily provide high test adequacy.

In Figure 10, both random 1% sample and first 1% sample provide 59% of what is covered by BigTest. We perform another experiment to measure the impact of different sample sizes on JDU path coverage and test execution time. Figure 11a and Figure 11b present the results on ComuteType. In ComuteType, the covered JDU paths increases from two to six when the percentage of the selected data increases from 0.1% to 50%. For those small samples, input tables do not have matching keys to exercise downstream operators and the time and distance columns may not have specific values to exercise all internal paths of the UDF. In terms of running time, as the sample size ($k$) increases, the test execution time also increases linearly (see Figure 11b in which x-axis is in log scale).

5.3 Fault Detection Capability

We evaluate BigTest’s ability to detect faults by manually injecting commonly occurring faults. Because DISC applications are rarely open-sourced for data privacy reasons and there is no existing benchmark of faulty DISC applications, we create a set of faulty DISC applications by studying the characteristics of real world DISC application bugs and injecting faults based on this study.

We carefully investigate Stack Overflow and Apache Spark Mailing lists with keywords; Apache Spark exceptions, task errors, failures, and wrong outputs and inspect top 50 posts. Many errors are related to performance and configuration errors; thus, we filter out those and analyze 23 posts related to coding errors. For each post, we investigate the type of fault by reading the question, posted code, error logs, answers, and accepted solutions. We categorize our findings into seven common fault types:

1. incorrect string offset: e.g., a user uses 1 instead of 0 as the starting index in method substring and encounters StringIndexOutOfBoundsException [7].
2. incorrect column selection: e.g., a user accesses a wrong column in a csv file and thus receives ArrayIndexOutOfBoundsException [5].
3. use of wrong delimiters: e.g., while splitting a string a user uses "[" instead of "["]", leading to a wrong output [8].
4. incorrect branch conditions: e.g., a user places a wrong order of control predicates, executing only one branch’s side [4].

As another example, application P6 identifies courses with more than 5 failing students. A faulty version of P6 replaces the filter predicate count>5 to count>0 to output courses with at least one failing student. The original version of P6 uses map and filter to parse each row and identify failing students, reduceByKey to count the number of failing students, and uses filter to find courses with more than 5 failing students. BigTest generates at least two records to exercise both terminating and non-terminating cases of the last filter; thus, the original and faulty versions produce different outcomes on this data. On the other hand, a record is generated to exercise a non-terminating case only. Such data

Table 4: Fault detection capabilities of BigTest and Sedge

<table>
<thead>
<tr>
<th>Seeded Faults</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detected by BigTest</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Detected by Sedge</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5: Modelling terminating and non-terminating cases

<table>
<thead>
<tr>
<th>Approach</th>
<th>Test Input Data</th>
<th>Output from Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigTest Terminating</td>
<td>CS100:31500</td>
<td>CS200:0.0,0.0,0.0,0.0</td>
</tr>
<tr>
<td>Non-terminating</td>
<td>CS100:0</td>
<td>CS200:0</td>
</tr>
<tr>
<td>Sedge Non-terminating</td>
<td>CS200:200.0,0.0,0.0,0.0</td>
<td></td>
</tr>
<tr>
<td>Terminating</td>
<td>CS100:0</td>
<td>CS200:0</td>
</tr>
</tbody>
</table>
would produce the same outcome for both the original and the faulty versions, unable to detect the injected fault, as shown in Table 5.

### 5.4 Testing Data Reduction

Testing DISC applications on the entire dataset is expensive and time-consuming. **BigTest** minimizes the size of the dataset, while maintaining the same test coverage. It generates only a few data records (in order of tens) to achieve the same JDU path coverage as the entire production data. Four out of seven benchmarks have an accompanied dataset, whereas the rest relies on a synthetic dataset of around 20GB each. Figure 12 shows the comparison result. In application P6, **BigTest** generates 30 rows of data to achieve 33% more JDU path coverage than the entire dataset of 40 million records. In other words, **BigTest** produces testing data 10^6 times smaller than the original dataset. Across all benchmark applications, **BigTest** generates data ranging from 5 to 30 rows. This is 10^2 to 10^6 times smaller than the original dataset, showing the potential to significantly reduce dataset size for local testing.

### 5.5 Time and Resource Saving

By minimizing test data without compromising JDU path coverage, **BigTest** consequently reduces the test running time. The benefit of a smaller test data is twofolds: (1) the amount of time required to run a test case decreases, and (2) the amount of resources (worker nodes, memory, disk space, etc.) for running tests also decreases.

We measure, on a single machine, the total running time by **BigTest** and compare it with the testing time on a 16-node cluster with the entire input dataset. We present a breakdown of the total running time into test data generation vs. executing an application on the generated data. Figure 13 represents the evaluation results. In application P6, it takes 5.3 seconds on a single machine to test with data from **BigTest** otherwise testing takes 387.2 CPU seconds (24.2 seconds x 16 machines) on the entire dataset, which still lacks complete JDU path coverage. Across the seven subject programs, **BigTest** improves the testing time by 194X, on average, compared to testing with the entire dataset.

### 5.6 Bounded Depth Exploration for Aggregation

**BigTest** takes a user-provided bound K to bound the number of times a loop is unrolled. We assess the impact of varying K from 1 to 5 and present the results in Figure 15. At K=2, the number of JDU paths for **GradeAnalysis** is 36. When K=3, **BigTest** generates 438 JDU paths. An exponential-like increase in the test generation time can be seen across the subject program, as we increase K. When K=2 in **GradeAnalysis**, **BigTest** takes 12 seconds and with K=3, **BigTest** takes 204 seconds. We empirically find K=2 to be a reasonable upper bound for loop iteration to avoid path explosion.

### 5.7 Threats to Validity

As we manually seed faults in the benchmark applications, the location of faults may introduce a bias in fault detection rate of **BigTest** posing a threat to internal validity. However, as mentioned before, most type of faults are only applicable to a single code location. If a fault type is applicable to multiple locations, we then select the fault location inspired by the corresponding StackOverflow/Mailing List post. In case of external validity, our classification of DISC faults may not be representative of all possible DISC application faults out there, as the survey is based on 50 StackOverflow/mailing lists posts. Additionally, the selection of fault types in our evaluation may be unfair to prior approaches. We attempt to mitigate this bias by restricting the evaluation to top seven most commonly occurring
faults in DISC applications. To eliminate this threat in the future, we plan to perform a large scale study on DISC application faults.

6 RELATED WORK

Testing Map-Reduce Programs. Csilaner et al. propose the idea of testing commutative and associative properties of Map-Reduce programs by generating symbolic constraints [18]. Their goal is to identify non-determinism in a Map-Reduce program arising from a non-associative or non-commutative user-defined function in the reduce operator. They produce counter examples as evidence by running a constraint solver over symbolic path constraints. Xu et al. add few more Map-Reduce program properties such as (1) operator selectivity, (2) operator statefulness, and (3) partition interference [46]. Both of these techniques test only high-level properties of individual dataflow operators and they do not model the internal program paths of user-defined functions. Olsten et al. generate data for Pig Latin programs [34]. Compared to random sampling, their approach provides concise data selected from an input dataset to achieve better statement coverage. Their approach considers each operator in isolation and does not model internal program paths of UDFs—treated as black-box. Furthermore, Olsten et al. require knowing the inverse function of a UDF given to transform.

Li et al. (SEDE) [31] is the most relevant approach to BigTest. SEDGE has three main limitations. First, its symbolic execution does not analyze the internal paths of individual UDFs. It considers UDFs as black box procedures and encodes them into uninterpreted functions. Second, it does not support operators such as flatMap, reduce, and reduceByKey, which are essential for constructing a collection and aggregating results from a collection in big data analytics. Third, the equivalence class modeling for each dataflow operator is not comprehensive, as it does not consider early terminating cases for some operators, where a data record does not flow to the next dataflow operator. Our empirical evaluation in Section 5 finds that these limitations lead to low defect detection in SEDGE. Table 6 compares dataflow operator support for related approaches and shows that BigTest has the most comprehensive and advanced support for modern DISC applications.

Test Generation in Databases. JDBC [42] or ODBC [2] enable software developers to write applications that construct and execute database queries at runtime. Testing such programs requires test inputs and database states from a user. Emmi et al. perform concolic execution of a program embedded with an SQL query [21] by symbolically executing the program till the point where a query is executed. Their approach is only applicable to basic SQL operations such as projection, selection, etc. (e.g., SELECT...FROM...WHERE). Pan et al. perform database state generation but with different coverage criteria goals [36]. Braberman et al. select input data to test the logic of computing additional fields from existing columns in the database [13]. They do not handle arbitrary user-defined functions which are prevalent in DISC applications.

Symbolic Execution. Symbolic execution is a widely used technique in software engineering [12, 27, 38] and is used to generate test data using constraint solvers [14–16, 23, 32, 33, 41]. For example, Visser et al. use JPF (Java PathFinder [29]) to generate test input data [45]. However, the same approach cannot be applied to DISC applications directly because it would symbolically execute the application as well as the underlying DISC framework. Such practice will produce an unnecessarily large number of complex path constraints, facing scalability issues. This justifies and motivates our approach that abstracts each dataflow operator as a logical specification while performing symbolic execution for each UDF.

Rosette is a framework for designing a solver-aided language [43] to ease the process of translating each language construct into symbolic constraints. BigTest and Rosette share some similarities in that they both translate higher-order types such as tuples or arrays into lower-level constraints. For the purpose of side-channel analysis, Bang et al. address a similar problem of solving constraints that cross boundaries between different theories (numerics, integer, and string constraints) [10]. Such cross-theory constraints are known to be difficult to solve with Z3 or CVC4. They extend SPf by modeling strings into bit vectors and by integrating numeric model counting in ABC [9] which could be used for BigTest in the future.

Regression Testing. Regression testing has been extensively studied in software testing. Safe regression testing selects only those test cases that exercise the updated regions of a program [26]. Rothermel et al. summarize several regression testing techniques and evaluate them under a controlled environment [39]. Test augmentation techniques help developers generate new test data to cover code not exercised by the available test cases using symbolic execution [17, 30]. Xu et al. evaluate concolic and genetic test generation approaches and report trade-offs [47]. The aforementioned approaches are not directly applicable to DISC applications, as they do not explicitly model the combined behavior of dataflow (relational) operators and the internal semantics of UDFs.

7 CONCLUSION AND FUTURE WORK

Big data analytics are now prevalent in many domains. However, software engineering methods for DISC applications are relatively under-developed. To enable efficient and effective testing of big data analytics in real world settings, we present a novel white-box testing technique that systematically explores the combined behavior of dataflow operators and corresponding user-defined functions. This technique generates joint dataflow and UDF path constraints and leverages theorem solvers to generate concrete test inputs.

BigTest can detect 2X more faults than the previous approach and can consume 194X less CPU time, on average than using the entire dataset. With BigTest, fast local testing is feasible and testing DISC applications on the entire dataset may not be necessary.

Acknowledgments. We thank the anonymous reviewers for their comments. The participants of this research are in part supported by Google PhD Fellowship, NSF grants CCF-1764077, CCF-1527923, CCF-1460325, CCF-1723773, ONR grant N00014-18-1-2037, Intel CAPA grant, and Samsung grant. We would also like to thank Emina Torlak and Koushik Sen for their insightful discussions.

<table>
<thead>
<tr>
<th>Dataflow Operators</th>
<th>Olsten et al.</th>
<th>Li et al.</th>
<th>Emmi et al.</th>
<th>Pan et al.</th>
<th>BigTest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map (Select)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Map (transform)</td>
<td>Incomplete</td>
<td>Incomplete</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Filter (Where)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Group</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Join</td>
<td>Incomplete</td>
<td>Incomplete</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FlatMap (Split)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Intersection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reduce</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6: Support of dataflow operators in related work
PUBLICATIONS


...


