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 inadequate 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 10^9 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 [48] are commonly used today to process terabytes and petabytes of data. At this scale, rare and buggy corner cases frequently show up in production [49]. 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, 39]. 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 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 (satisfiability modulo theories) 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 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 BigStitt [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 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—BigTest reveals 2X more manually injected faults than Sedge on average. BigTest can minimize data size for local testing by 10^3 to 10^8 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.
- It 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.

---

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

---

**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 the paper.

## 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 [47]. 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) [36]. Internally, SPF relies on the
To map an area zip code to its corresponding area name, Alice uses another dataset that assigns a name to certain zip codes. For example, the two data sets (line 13) in the subsequent analysis engine of Java PathFinder (JPF) model checking [43]. 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 \( z = x/y \) 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 the trip, the start and end location in terms of a zip code, the distance in miles, and the trip duration in hours, for example, 39034, Culver City. 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: 39034, 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 \( \theta \) 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 map \( \theta \) at lines 15-18. There are three internal path conditions: (1) speed \( > 40 \) mph, (2) \( 15 \) mph \( < \) speed \( \leq 40 \) mph, and (3) speed \( \leq 15 \) mph. The sub figure \( \phi \) in Figure 3b shows corresponding path conditions and effects.

**String Constraints.** To analyze the second map \( \psi \) 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 \( Z . \text{split}(\"\"\}). \text{length} \geq 2 \) 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.
Otherwise, this application may crash in production, when the input record does not have an expected delimiter.

**Arrays.** To analyze reduceByKey at line 20 (also in Figure 3b), we must model how the UDF operates on the input array of size K, \([a_1, a_2, \ldots, a_K]\) and produces the corresponding output \(f_{agg}(a_1, f_{agg}(a_2, \ldots, f_{agg}(a_{K-1}, a_K)) \ldots))\). For example, the UDF \(_+_\) returns the sum of two input arguments. When the array size K is given by a user, the final output \(N= a_1 + (a_2 + \ldots (a_{K-1} + a_K))\).

**Summary.** Due to the internal path conditions entailed by individual UDFs, instead of four high-level dataflow paths shown in Figure 3a, Alice must consider eleven paths in total, which are enumerated in Table 1. Figure 3 shows the symbolic execution tree at the level of dataflow operators on the left and the internal symbolic execution trees for individual UDFs on the right. Lastly, example data generated by BigTest for each JDU path using Z3 is shown in Table 1. While these example data records may not look realistic, such data is necessary to exercise the downstream UDFs that are otherwise unreachable with the original dataset. For instance, filtering a dataset without any passing data record will result in an empty set and consequently, the UDFs after the filter will never get tested with the original data. Therefore, synthetic data is necessary and crucial to expose downstream program behavior.

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

<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 &lt; 5</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>C4</td>
<td>T.split(',').length ≥ 5 ∧ isInt(T.split(',')(3)) ∧ isInt(T.split(',')(4))</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>C4</td>
<td>T.split(',').length ≥ 5 ∧ isInt(T.split(',')(4)) ∧ T.split(',')(4).toInt = 0</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>C5</td>
<td>T.split(',').length &lt; 2</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>C6</td>
<td>Z.split(',').length ≥ 2 ∧ a_2 = &quot;Palms&quot;</td>
<td>-1</td>
<td>&quot;\text{\texttt{\textit{C10}}}&quot;, &quot;\text{\texttt{\textit{C11}}}&quot;, &quot;\text{\texttt{\textit{C6}}}&quot;</td>
</tr>
<tr>
<td>C7</td>
<td>T.split(',').length ≥ 5 ∧ isInt(T.split(',')(3)) ∧ isInt(T.split(',')(4)) ∧ T.split(',')(4).toInt = 0</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>C8</td>
<td>T.split(',')(4).toInt = 0</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>C9</td>
<td>Z.split(',').length ≥ 2</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>C10</td>
<td>V.split(',').length ≥ 2 ∧ a_2 = &quot;Palms&quot;</td>
<td>-1</td>
<td>&quot;\text{\texttt{\textit{C10}}}&quot;, &quot;\text{\texttt{\textit{C11}}}&quot;, &quot;\text{\texttt{\textit{C6}}}&quot;</td>
</tr>
<tr>
<td>C11</td>
<td>T.split(',')(4).toInt = 0</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

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.

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.

```scala
1. def f(a:Int,b:Int){
2.   return a+b;
3. }
4. //Usage in reduce
5. reduce(reduce(f));
6. return sum;
```

(a) A normal invocation of reduce with a corresponding UDF.

(b) An equivalent iterative version with a bound K

**Handling Aggregator Logic.** For aggregation operators, the attached UDF must be transformed. For example, the UDF for reduce is an 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).

**4.2 Logical Specifications of Dataflow Operators**

This section describes the equivalence classes generated by each dataflow operator’s semantics. We use \(C_I\) to represent a set of path constraints on the input data, \(I\), for a particular operator. A single element \(c\) in \(C_I\) contains path constraints that must be satisfied to execute a corresponding unique path. We define \(f\) as the set
of symbolic path constraints of a UDF where \( f(t) \) represents constraints of a unique path exercised by input \( t \). By abstracting the implementation of dataflow operators into logical specifications, BioTestr does not have to symbolically execute the Spark framework code (about 700KLOC), as it focuses on application level faults only as opposed to framework implementation faults which is out of the scope of this paper. BioTestr supports all popular dataflow operators with the exception of deprecated operators such as co-join.

**Filter.** Filter takes a boolean function \( udf \) deciding if an input record should be passed to downstream operators or not. Therefore, we model two equivalence classes: (1) there exists a record \( t \) satisfying one of the incoming constraints, \( C_I \) from input table \( I \) (i.e., the table produced by its upstream satisfying operator), shown in (1) Table 2; (2) there exists a record \( t \) that satisfies one of the incoming constraints but not \( udf \), shown in (2) Table 2.

**Map and Flatmap.** Map takes a UDF \( udf \) as an input and applies it to each input record to produce an output record. It has one equivalence class, where there exists tuple \( t \) from the input table \( I \) satisfying one of the incoming constraints, \( c \in C_I \) and also one of the path constraints in \( f \) i.e., path constraints generated by symbolically executing \( udf \), shown in (3) Table 2. Map is supported by the previous work SEdge but SEdge considers the UDF \( udf \) as a black box, uninterpreted function. Flatmap splits an input record using a \( udf \) to generate a set of records, and thus the equivalence class of flatmap is similar to that of map, as shown in (4). BioTestr handles flatmap by explicitly modeling a collection, described in Section 4.3.

**Join.** Join performs an inner-join of two tables \( t_r \) on the right and \( t_l \) on the left based on the equality of keys, assuming that records from both tables are of the type Tuple (key, value). We model the output records of join into three equivalence classes: (1) the key of tuple \( t_g \) in the right table matches with a key of tuple \( t_l \) on the left; (2) the key of tuple \( t_g \) in the right table does not match with any key of tuple \( t_l \) on the left; and (3) the key of tuple \( t_g \) in the left table does not match with any key of tuple \( t_l \) on the right. (1), (2), and (3) in Table 2 represent the three equivalence classes.

**Reduce and ReduceByKey.** Reduce takes a \( udf \) and a collection as inputs and outputs an aggregated value, while reduceByKey performs a similar operation per key. As discussed in Section 4.1, BioTestr generates an equivalent iterative version of the \( udf \) with a loop. By this refactoring of \( udf \) to \( udf' \), the equivalence classes could be modeled similar to that of map, where there exist input records \( t_1, t_2, \ldots, t_n \in I \) on which each of the corresponding non-terminating constraint \( (c_1, c_2, \ldots, c_n) \in C_I \) from the input table \( I \) holds true. In addition, each record must satisfy the constraints of \( udf' \), satisfying \( f'((t_1, t_2, \ldots, t_n)) \), as shown in (6) of Table 2.

### 4.3 Path Constraint Generation

This section describes several enhancements in Symbolic Path Finder (SPF) to tailor symbolic execution for DISC applications. DISC applications extensively use string manipulation operations and rely on a Tuple data structure to enable key-value based operations. Using an off-the-shelf SPF naively on a UDF would not produce meaningful path conditions, thus, overlooking faults during testing.

1. def parse(s:String)
2. val cols = s.split(“,”)
3. (cols(0), vals(cols(1)))

**Figure 6:** A UDF with string manipulation

**Strings.** Operations such as split for converting an input record into a key-value pair are common in DISC applications but are not supported by SPF. BioTestr extends SPF by capturing calls to split, recording the delimiter, and returning an array of symbolic strings. When an \( n \)-th element of this symbolic array is requested, SPF returns a symbolic string encoded as splitn with a corresponding index. By representing the effect of Figure 6 as (splitn(“,”, s, \( n \)), \( n \)), SPF generates one terminating constraint, where \( s \) can only split into fewer than two segments, and one non-terminating constraint where \( s \) can split into at least two segments. Due to no split support, naïve SPF generates a string without any delimiter as a test input e.g., “x000" instead of "x000,000". This input would lead to ArrayIndexOutOfBoundsException while accessing a string using split(",”,1).

1. def agg(arr:Array[Int])
2. val sum = arr(0); // Bound K=3
3. for(a <- 1 to min(arr.size,2)) sum += arr(a)@0 ? 0 : arr(a)
4. sum

**Figure 7:** An iterative version of aggregator UDF
produced by BigTest takes a collection as input and sums up each element, if the element is greater than or equal to zero. Given a user-provided bound $K=3$ BigTest unrolls the loop three times and generates four pairs of a path condition (P) and the corresponding effect (E):

1. $P:a(1)<0 \land a(2)<0, E:a(0)$
2. $P:a(1)\geq 0 \land a(2)<0, E:a(0)\lor a(1)$
3. $P:a(1)<0 \land a(2)\geq 0, E:a(0)\lor a(2)$
4. $P:a(1)\geq 0 \land a(2)\geq 0, E:a(0)\lor a(1)+a(2)$

A naïve SPF does not handle collections well and thus may generate an array of length 1 only, not exercising line 3 in Figure 7. For example, `agg((3))` outputs the same sum of 3, when $arr(a)<0$ is mutated to $arr(a)>0$, because the loop starts from 1 instead of 0, and sum is initialized to the first element of the array. Thus, it is not possible to detect the fault using an array of length 1.

**Exceptions.** BigTest extends SPF to explicitly model exceptions. For example, when an expression involves a division operator, division by zero is possible, which can lead to program termination. In Figure 1, BigTest creates two additional terminating path conditions, due to division by zero (i.e., $x*y/x*y=0$ and $y/x*y=0$).

**Combining UDF symbolic execution with equivalence classes.** BigTest combines the path conditions of each UDF with the incoming constraints from its upstream operator. For example, the UDF of `filter()` in Section 3 produces a path condition of $\text{splitn}"\),"\).length\geq 2$ with the effect $\text{splitn}(\"","\),1)$. Inside the equivalence classes of filter—rows $0$ and $1$ in Table 2, BigTest plugs in the incoming path conditions (effects) of an upstream operator map to $C_1$ and the path conditions (effects) of the filter’s UDF to $f$, producing the following path conditions.

- $c(t) \land f(t): \text{splitn}(\"",\).length\geq 2 \land \text{splitn}(\"",\),1) == \"Palms\"$
- $\neg (c(t) \land \neg f(t)): \text{splitn}(\"",\).length\geq 2 \land \text{splitn}(\"",\),1) == \"Palms\"$

**Joint Dataflow and UDF Path.** BigTest defines the final set of paths of a DISC application as **Joint Dataflow and UDF (JDU) paths**. We define a JDU path as follows: let $G = (D,E)$ represent a directed acyclic graph of a DISC application where $D$ is a set of vertices representing dataflow operators and $E$ represents directed edges connecting dataflow operators. Imagine a DISC application constructed with a map followed by filter and reduce. We represent this dataflow graph as $G = (D,E)$ such that $D = \{d_1,d_2,d_3,t_1\}$ and $E = \{(d_1,d_2),(d_2,d_3),(d_3,t_1)\}$ where $d_1$, $d_2$, and $d_3$ are map, filter, and reduce respectively. BigTest introduces a terminating edge $(d_3,t_1)$ where a terminating vertex $t_1$ is.

Since each dataflow operator takes a user-defined function $f$, for a vertex $d_i$, we define a subgraph $G_i = (V_i,E_i)$ which represents the control flow graph of $f$. In this subgraph, a vertex $v \in V_i$ represents a program point and an edge $(v_a,v_b) \in E_i$ represents the flow of control from vertex $v_a$ to $v_b$. $G_i$ has $v_1 = \text{start}$ and $v_n = \text{stop}$ corresponding to the first and last statements. Then from each dataflow operator node $d_i$, we add a call edge from $d_i$ to the start node of $G_i$ and from the stop node of $G_i$ to the $d_{i+1}$. Since some UDFs include a loop and thus have a cycle in the control flow graph, we enumerate a set of all unique paths $P_G$ for the graph $G$ with expanded subgraphs and call each unique path a **Joint Dataflow and UDF (JDU)** path. For an arbitrary test suite $T$, the JDU path coverage is measured as a set of covered paths, $P_K(T) = \{p \mid p \in P_G, \exists \in T$ and $t \in C_p\}$ where a test input $t$ satisfies the path condition $C_p$ of path $p$. Given a user-provided bound $K$ for unrolling a loop, JDU path coverage is $P_K(T)/|P_G|$. The next step is to finitize the loop using a user provided bound $K$ and unroll the loop $K$ times.

We enumerate a set of all unique paths $P_K$ for the graph $G$ with expanded subgraphs and call each unique path a **Joint Dataflow and UDF (JDU)** path. For an arbitrary test suite $T$, the JDU path coverage is measured as a set of covered paths, $P_K(T) = \{p \mid p \in P_G, \exists \in T$ and $t \in C_p\}$ where a test input $t$ satisfies the path condition $C_p$ of path $p$. Given a user-provided bound $K$ for unrolling a loop, JDU path coverage is $P_K(T)/|P_G|$. The next step is to finitize the loop using a user provided bound $K$ and unroll the loop $K$ times.

Finally, we enumerate a set of all unique paths $P_K$ for the graph $G$ with expanded subgraphs and call each unique path a **Joint Dataflow and UDF (JDU)** path. For an arbitrary test suite $T$, the JDU path coverage is measured as a set of covered paths, $P_K(T) = \{p \mid p \in P_G, \exists \in T$ and $t \in C_p\}$ where a test input $t$ satisfies the path condition $C_p$ of path $p$. Given a user-provided bound $K$ for unrolling a loop, JDU path coverage is $P_K(T)/|P_G|$. The next step is to finitize the loop using a user provided bound $K$ and unroll the loop $K$ times.
that crosses the boundary of strings and integers, BigTest uses a custom function isInteger and Z3 function str. to. int. Lines 11 to 14 enforce a record to contain “Pals” 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 can BigTest be 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 load As and split. Three programs MovieRating (P2), AirportLayover (P3), and WordCount (P7) are adapted from BigShift [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 arithmetic, (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 storage 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 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 implementations 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 would be expected)

---

1. https://github.com/maliguzlar/BigTest
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 inject one of each fault type in every applicable example, fault types 1 and 3 could only be inserted when subStr or split method is used. When a fault type is applicable to multiple locations, we select a location which is inspired by and similar to the fault location in the corresponding StackOverflow/Mailing List post. For instance, for fault type (2) above, we manually modify code to extract the first column instead of the second as a key in line 4 of Figure 2. Similarly, for fault type (3), we introduce fault by replacing the delimiter “,” with “.”. In total, our benchmark comprises of 31 faulty DISC applications. While Sedge is not designed to handle string constraints, the main goal of this exercise is to justify the need to model UDFs and string constraints. BigTest represents the internal UDFs as uninterpreted functions and, therefore, is unable to model all internal UDF paths. Conversely, BigTest treats UDFs as interpreted functions by representing them symbolically and models all internal UDF paths (up to bound k) which is crucial for high coverage testing of UDF’s internal.

Table 4 shows a comparison of fault detection by BigTest and Sedge. BigTest detects 2x more injected faults than Sedge. For instance, in application P4, BigTest detects 6 faults, whereas Sedge detects 4 faults. Sedge uses concrete execution to model the UDF exercising line 16 of Figure 2 only. Therefore, it is unable to find an input for detecting fault at line 17 when the binary operator “>” is replaced with “<” (i.e., s._2._1>15 to s._2._1<15). Similarly, when join in line 13 is changed to rightOuterJoin, Sedge cannot detect any difference in the output because the equivalence classes do not model the terminating cases of join.

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
Minimal input data selected for maximal JDU coverage

<table>
<thead>
<tr>
<th>Program</th>
<th>$10^3$</th>
<th>$10^4$</th>
<th>$10^5$</th>
<th>$10^6$</th>
<th>$10^7$</th>
<th>$10^8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>4</td>
<td>5</td>
<td>5.27</td>
<td>4.48</td>
<td>3.22</td>
<td>2.47</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
<td>5</td>
<td>4.73</td>
<td>3.48</td>
<td>2.21</td>
<td>1.37</td>
</tr>
<tr>
<td>P3</td>
<td>4</td>
<td>5</td>
<td>4.96</td>
<td>3.62</td>
<td>2.41</td>
<td>1.61</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>5</td>
<td>4.87</td>
<td>3.62</td>
<td>2.37</td>
<td>1.58</td>
</tr>
<tr>
<td>P5</td>
<td>4</td>
<td>5</td>
<td>4.87</td>
<td>3.62</td>
<td>2.37</td>
<td>1.58</td>
</tr>
<tr>
<td>P6</td>
<td>4</td>
<td>5</td>
<td>4.87</td>
<td>3.62</td>
<td>2.37</td>
<td>1.58</td>
</tr>
<tr>
<td>P7</td>
<td>4</td>
<td>5</td>
<td>4.87</td>
<td>3.62</td>
<td>2.37</td>
<td>1.58</td>
</tr>
</tbody>
</table>

Figure 12: Reduction in the size of the testing data by BigTest

Entire data
Test data generated by BigTest

<table>
<thead>
<tr>
<th>Program</th>
<th>Test Running Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>15.5</td>
</tr>
<tr>
<td>P2</td>
<td>13.7</td>
</tr>
<tr>
<td>P3</td>
<td>59.9</td>
</tr>
<tr>
<td>P4</td>
<td>12</td>
</tr>
<tr>
<td>P5</td>
<td>64.4</td>
</tr>
<tr>
<td>P6</td>
<td>9.7</td>
</tr>
<tr>
<td>P7</td>
<td>52.6</td>
</tr>
</tbody>
</table>

Figure 13: Test running time of entire data on large-scale cluster vs. testing on local machine with BigTest

outcomes on this data. On the other hand, a record is generated to exercise a non-terminating case only. Such data 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 $10^3$ to $10^5$ 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 test running 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,
we plan to perform a large scale study on DISC application faults. Testing Map-Reduce Programs. Csallner 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 [45]. 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]. 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 it.

Li et al. (SEDGE) [31] is the most relevant approach to BicTest. 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 BicTest has the most comprehensive and advanced support for modern DISC applications.

Test Generation in Databases. JDBC [41] 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, WHERE). Braberman et al. select input data to test the logic of computing additional fields such as projection, selection, etc. (e.g., SELECT, WHERE). Their approach is only applicable to basic SQL operations such as projection, selection, etc.

Symbolic Execution. Symbolic execution is a widely used technique in software engineering [12, 27, 37] and is used to generate test data using constraint solvers [14–16, 23, 32, 33, 40]. For example, Visser et al. use JPF (Java PathFinder [29]) to generate test input data [44]. 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 dataflow operators as a logical specifications while performing symbolic execution for the UDFs.

Rosette is a framework for designing a solver-aided language [42] to ease the process of translating each language construct into symbolic constraints. BicTest and Rosette both translate higher-order types such as arrays into lower-level constraints. Bang et al. address the problem of solving constraints crossing boundaries between different theories (numbers, 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 BicTest 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 [38]. 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 [46]. 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

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 UDFs. This technique generates joint dataflow and UDF path constraints and leverages theorem solvers to generate concrete test inputs. BicTest can detect 2X more faults than the previous approach and can consume 194X less CPU time, on average than using the entire dataset. With BicTest, 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.


