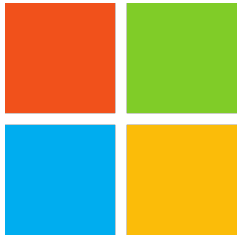
White-Box Testing of Big Data Analytics with Complex User-Defined Functions

Muhammad Ali Gulzar [1] Shaghayegh Mardani [1] **Madan Musuvathi[2]**
Miryung Kim[1]

[1]University of California, Los Angeles
[2]Mircrosoft Research



Inadequate Testing of Big Data Analytics

Software Development Cycle of Big Data Analytics

1. Develop locally

2. Test locally with **Sample Data**

3. Execute the job on the cloud hoping that it would work

4. **Sever hours later**, the job cashes or produces wrong output

5. Go to Step 2

Repeat

Motivating Example

Find the total number of trips made from UCLA using a public transport, a personal vehicle, or on foot.

Trips Dataset (20GB)

#,	ORIG,	DEST,	DIST,	TIME
1,	90034,	90024,	10,	1
2,	90001,	90024,	16,	1.4
….				

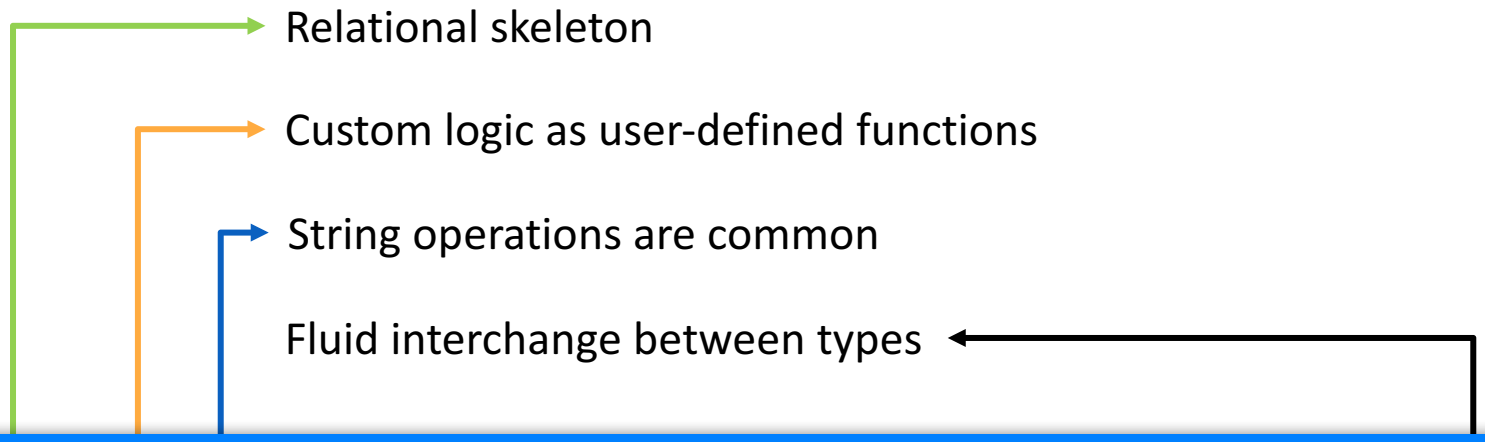
Locations Dataset (100MB)

Zip,	Location
90034,	“UCLA”
90024,	“Westwood”
…	

Big Data Application in Apache Spark

```
val trips = sc.textFile(“trips”)
  .map { s => val c = s.split(","); (c(1), c(3).toInt / c(4).toInt)}
val locations = sc.textFile(”zipcode”)
  .map { s => val c= s.split(","); (c(0), c(1))}
  .filter { s => s._2.equals(“UCLA") }
val result= trips.join(locations).map { s =>
       if (s._2._1 > 40) ("car", 1)
      else if (s._2._1 > 15) ("public", 1)
      else ("onfoot", 1)}
  .reduceByKey(_ + _)
```

Characteristics of Big Data Analytics



How do we test a big data application effectively and efficiently?

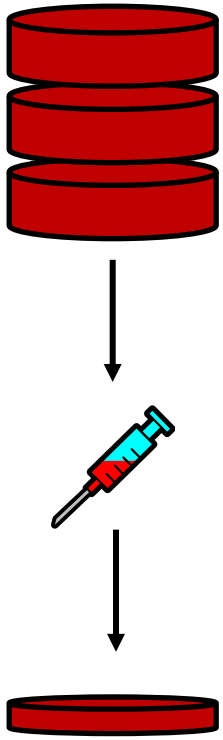
```
  .filter { s => s._2.equals("UCLA") }
val result= trips.join(locations).map { s =>
      if (s._2._1 > 40) ("car", 1)
     else if (s._2._1 > 15) ("public", 1)
     else ("onfoot", 1)}
  .reduceByKey(_ + _)
```

Option 1: Sample Input Data

- random sampling,
- top n sampling
- top k% sample, etc.

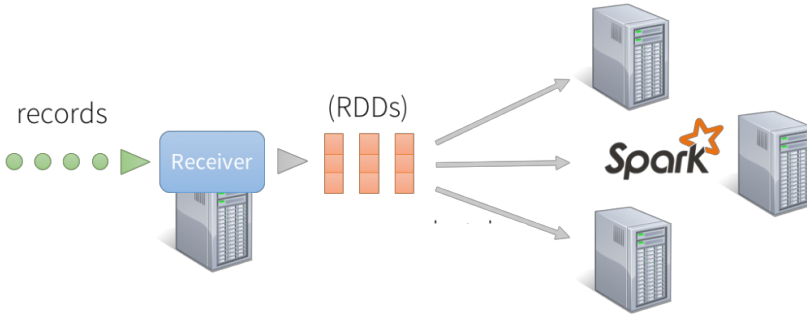
Limitations:

- The sample may only exercise a limited set of **program paths** (low code coverage).
- The sample may not include the inputs leading to a **program crash**.
- A large sample may have higher coverage but increase local **testing time**.



Option 2: Traditional Test Generation for Java

- Big Data Analytics programs compile to Java bytecode
- But this includes the entire system (700 KLOC for Apache Spark)



- **Symbolic execution** without abstraction is **infeasible** and would **not scale**

Our Approach: White-Box Testing

Input: Big Data Analytics Application

```
sc.textFile("hdfs")
.map(s=> s.toInt)
.filter(w => w > 0))
.reducebyKey(_+_)
```

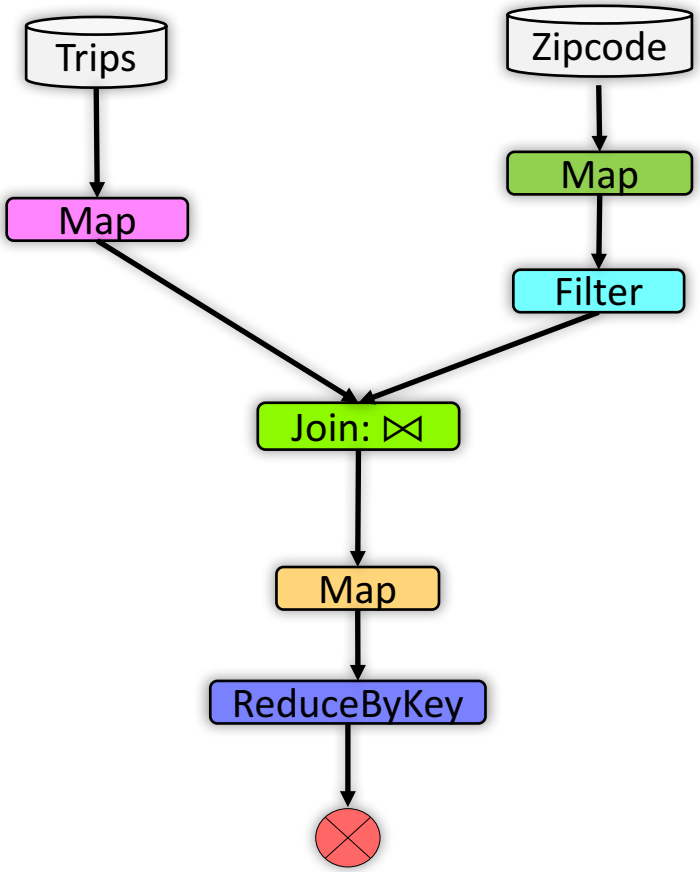
BigTest

Output: Test Input Data

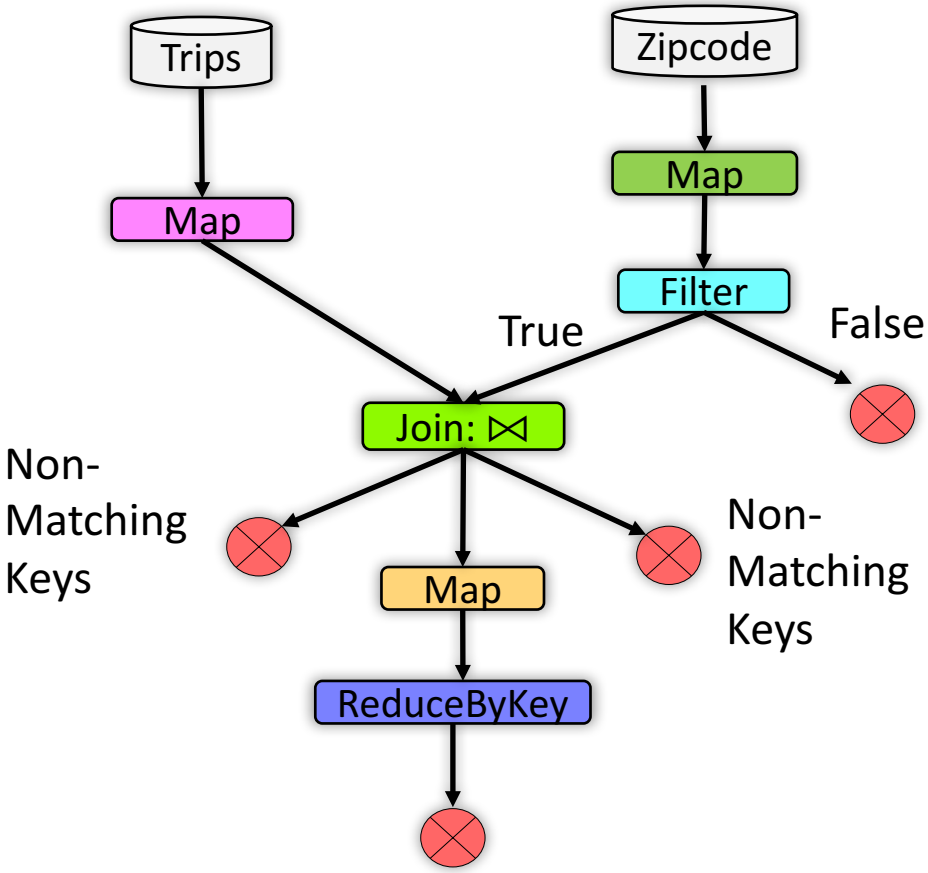
PC	Input
X>0	X="1"
X≤0	X="0"

1. Decompose relational skeleton and UDFs
2. Logical specifications for relational operators
3. Symbolic execution of UDFs
4. Generate inputs by joint path constraints

Modelling Dataflow Operators

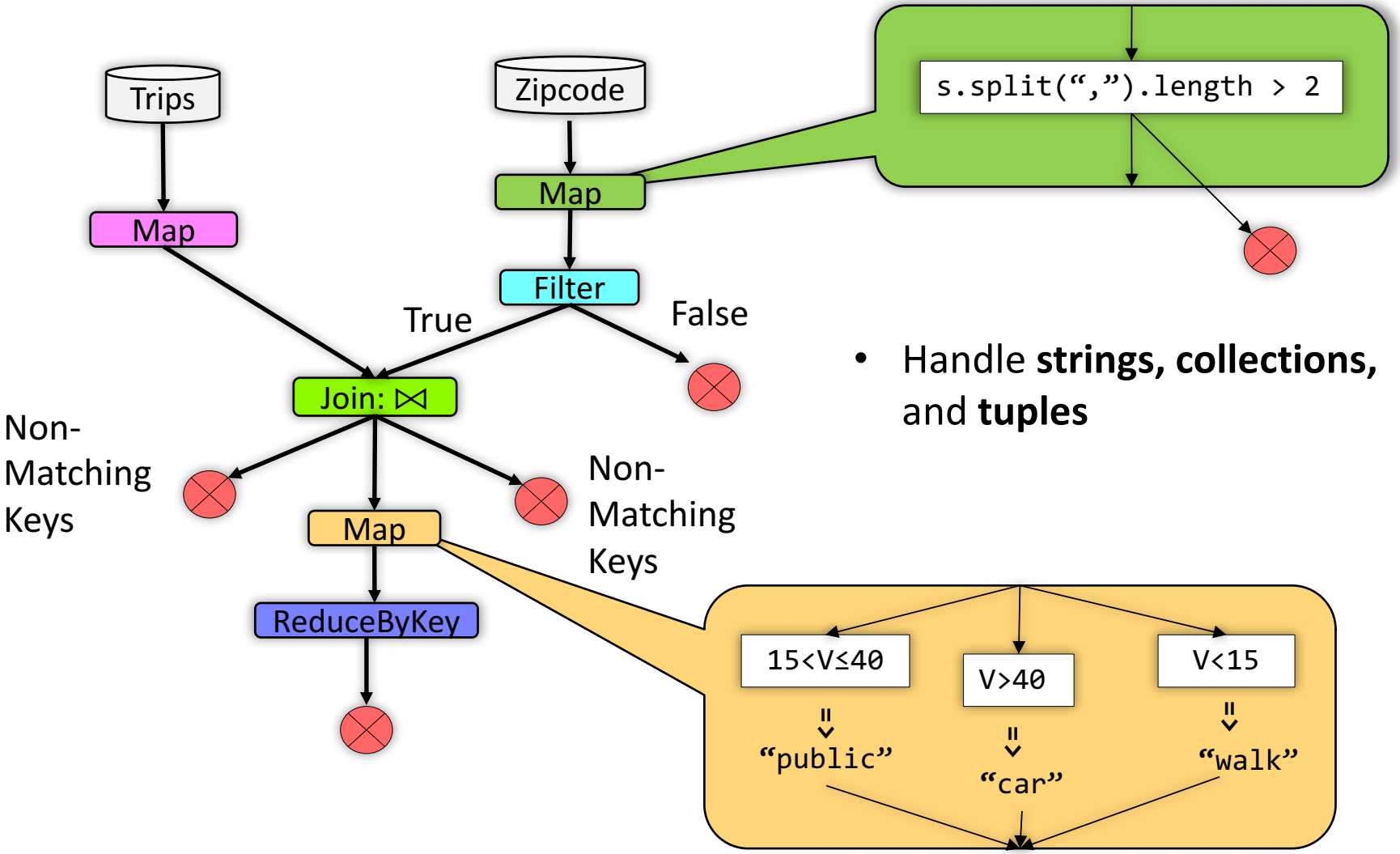


Modelling Dataflow Operators



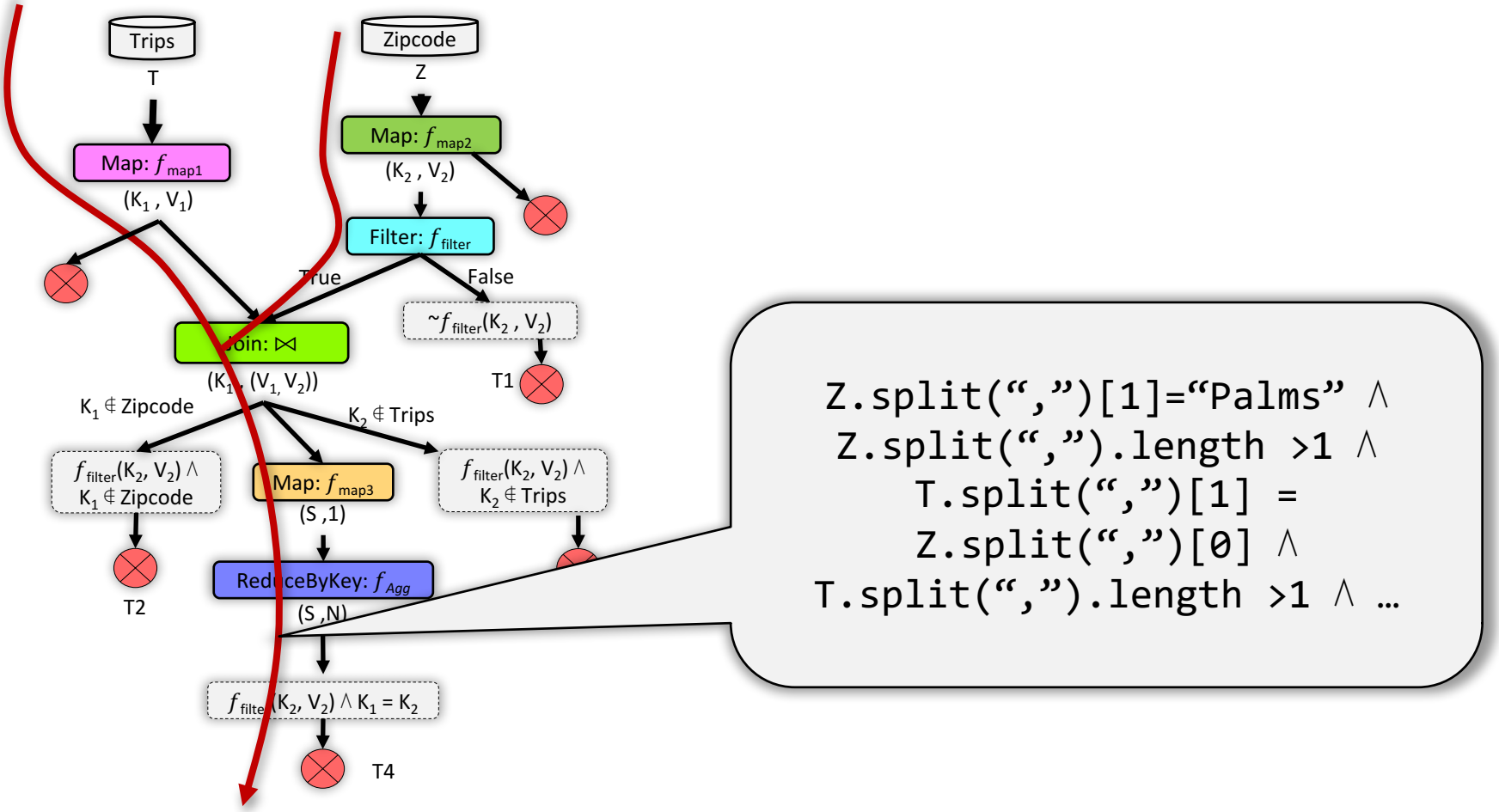
- Handle **terminating** and **non-terminating** cases of dataflow operators
- E.g. `Join` can introduce 3 cases
  - 2 cases in which keys from right and left do not match
  - 1 case in which right and left keys match

Modelling User-defined Functions



- Handle **strings, collections,** and **tuples**

Join Dataflow and UDF (JDU) Path



Test Input Generation

```
Z.split(“,”)[1]=“Palms” ∧
Z.split(“,”).length >1 ∧
   T.split(“,”)[1] =
   Z.split(“,”)[0] ∧
T.split(“,”).length >1 ∧ …
```

```
(assert (= T (str.++ (str.++ line20 ",") line21)))
(assert (= Z
           (str.++ (str.++ " " ",")
           (str.++ (str.++ line11 ",")
           (str.++ (str.++ " " ",") (str.++ (str.++ line13 ",")
           line14))))))
(assert
  (and (not (= (str.to.int line14) 0))
  (and (isinteger line14) (and (isinteger line13)
  (and (= "Palms" line21) (and (= x11 line20)
  (and (<= s21 15)
  (and (<= s21 40) (and (= s21 x621) (and (= s1 x61) (=
       s22 x622))))))))))))))
(assert
  (and (= x11 line11)
  (and (= x12 (/ (str.to.int line13) (str.to.int line14))) (and
(= x61 x11)
  (and (= x621 x12) (and (= x622 x42) (and (= x71 "walk") (= x72
1))))))))))))
```

Generated Test Data

Trips	Location
_, "\x00", _, "0", "1"	"\x00", "Palms"

Evaluation

RQ1: How much test coverage improvement can BigTest achieve?

RQ2: How many faults can BigTest detect?

- We built the first **benchmark of faulty dataflow programs** based on our survey of such programs on Q/A forums *e.g.* StackOverflow .

RQ3: How much test data reduction does BigTest provide and how long does BigTest take to generate test data?

Experimental Setting

- We use seven subject programs from earlier works
- All subject applications have complex string, complex arithmetic, Tuple type for key-value pairs, and collections with custom logic.

Subject Program	Dataflow Operators	# of Operators	JDU Paths K=2	# of UDFs
Income Aggregate	map, filter, reduce	3	6	4
Movie Ratings	map, filter, reduceByKey	4	5	4
Airport Layover	map, filter, reduceByKey	3	14	4
Commute Type	map, fitler, join, reduceByKey	6	11	5
PigMix-L2	map, join	5	4	6
Grade Analysis	flatmap, filter, reduceByKey, map	5	30	3
Word Count	flatmap, map, reduceByKey	3	4	3

Study of Big Data Analytics Faults

- No existing benchmark of faulty applications
- We study the characteristics of real-world big data analytics bugs posted on **StackOverflow** and **Apache Spark Mailing Lists**.

Survey Statistics	
Keywords Searched	Apache Spark exceptions, task errors, failures, wrong outputs
Posts Studied	Top 50
Posts with Coding Errors	23
Common Fault Types	7
Total Faulty Programs	31

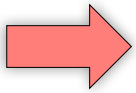
Fault Types	Example
Incorrect String Offset	`str.substring(1,0)`
Incorrect Column Selection	`str.split(“,”)[1]`
Wrong Delimiters	`str.split(“\t”)[1]`
Incorrect Branch Condition	`If(age>10 && age<9)`
Wrong Join Type	`LeftOuterJoin`
Key-Value Swap	`(Value, Key)`
Others	`Division by zero`

Real World Fault Injection

- Identified 7 common code fault types
- Manually inserted these faults into benchmarks
- Leads to a total of **31 faulty big data applications**.

Original Program

```
val trips = sc.textFile(“trips”)
 .map { s =>
   val c = s.split(",");
   (c(1), c(3).toInt / c(4).toInt)
}
val loc = sc.textFile(”zipcode”)
. . . .
```

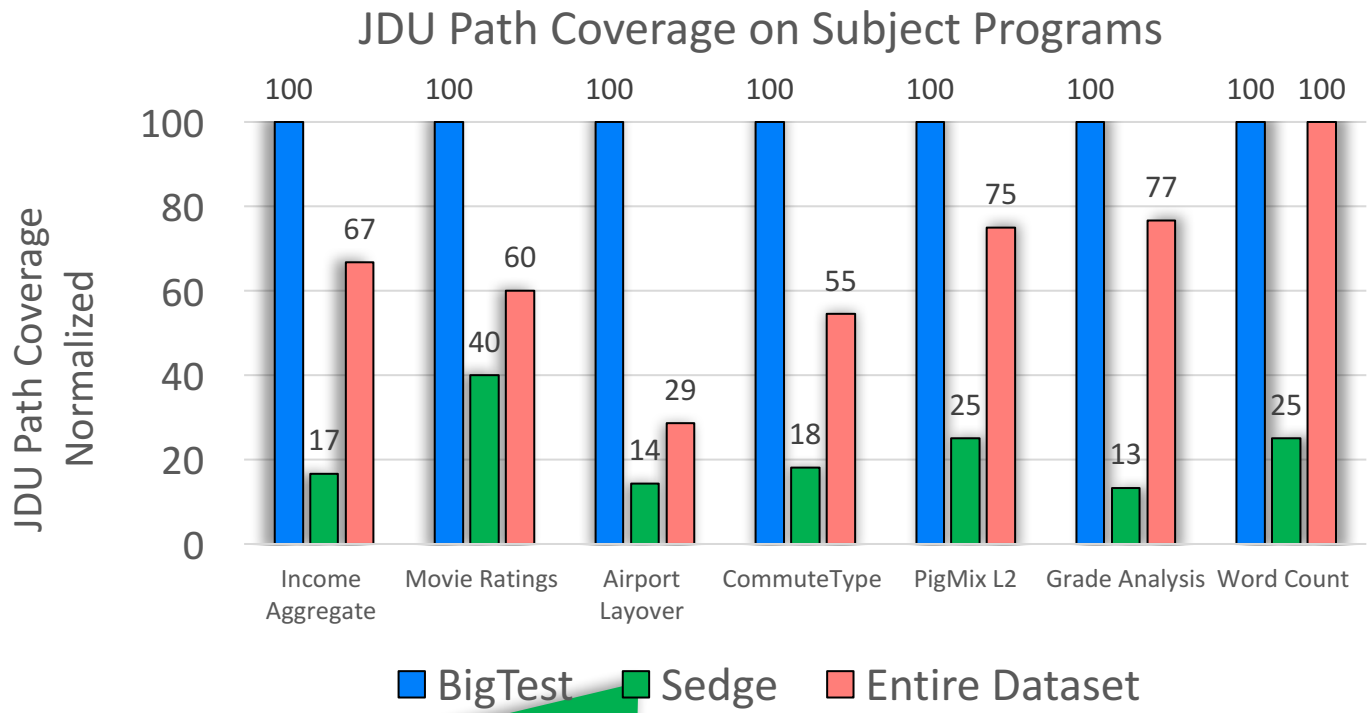


Faulty Program

```
val trips = sc.textFile(“trips”)
 .map { s =>
   val c = s.split(",");
-  (c(1), c(2).toInt / c(5). toInt)
}
val loc = sc.textFile(”zipcode”)
. . . .
```

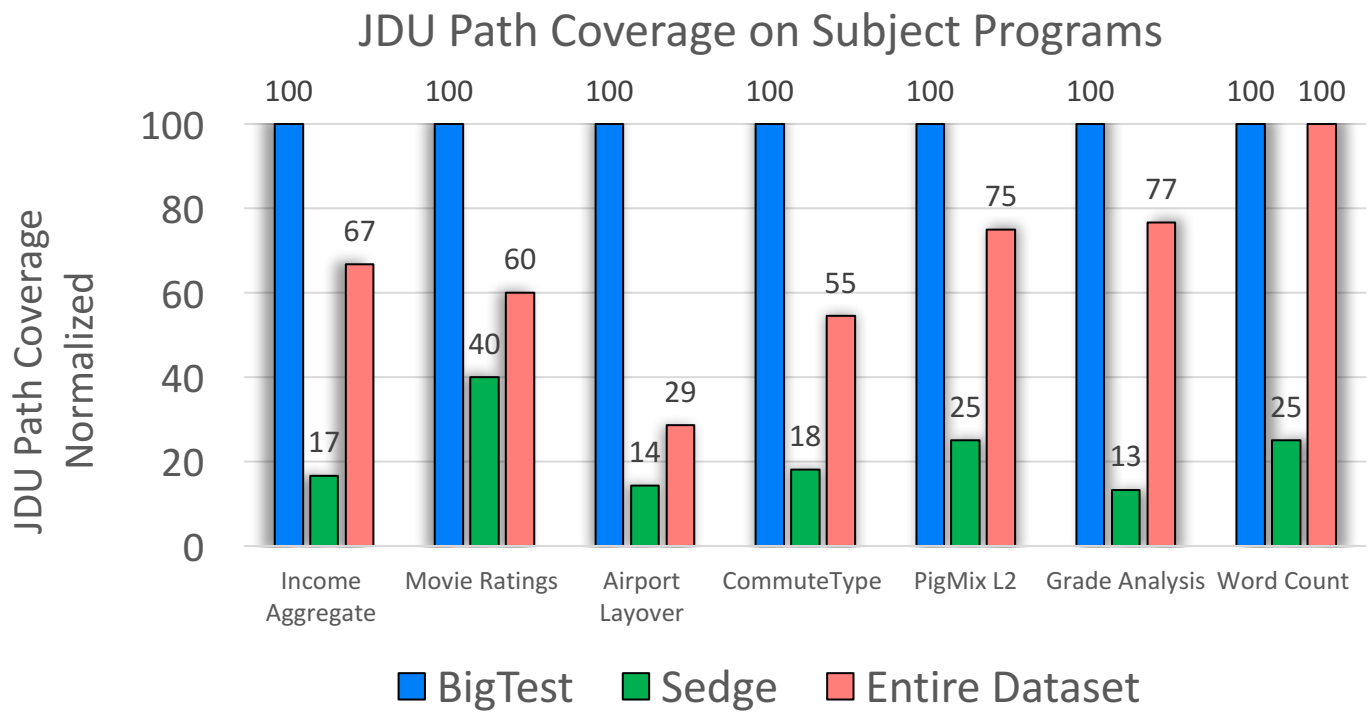
After injecting fault based on fault type *”Incorrect Column Selection”*, the program extracts the column at index 5 instead of 4.

RQ1: Code Coverage



Sedge [ASE '13] generates examples for dataflow programs but it handles a UDF as **uninterpreted** function and **does not model its internals**.

RQ1: Code Coverage



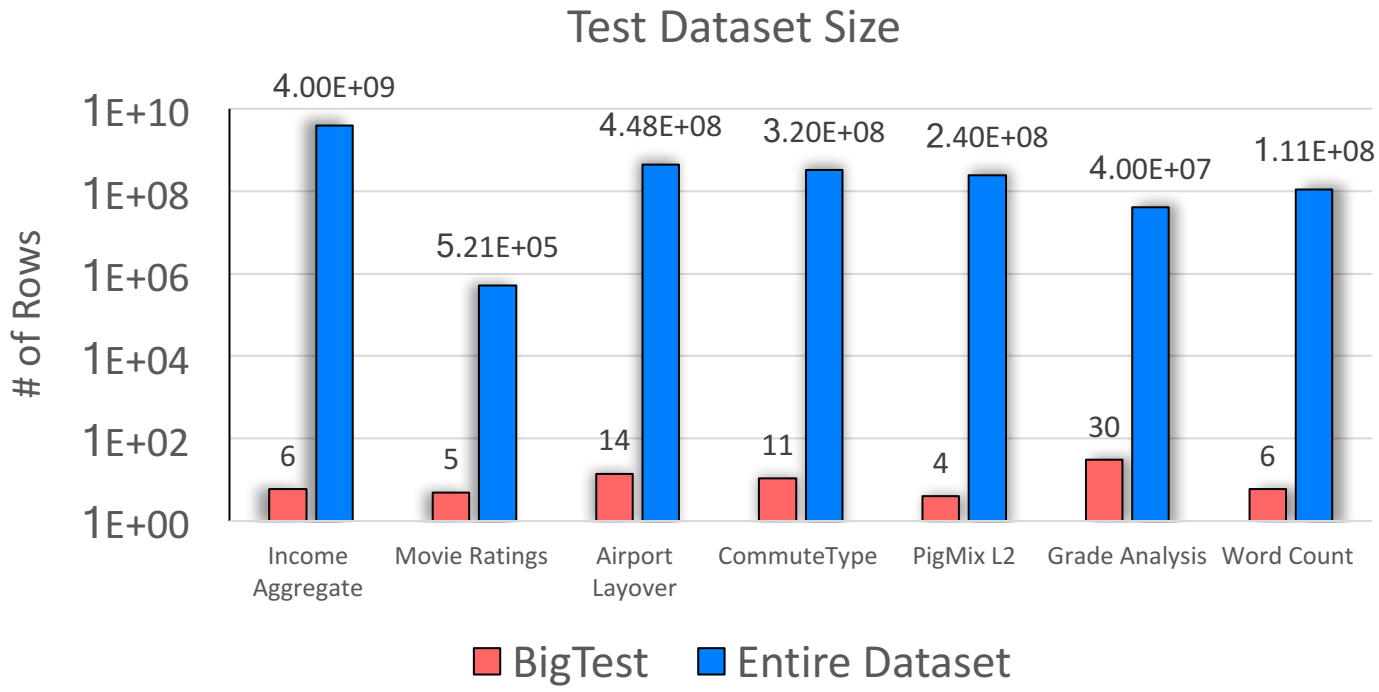
BigTest improves JDU path coverage by 78% against Sedge and 34% against the entire dataset.

RQ2: Fault Detection Capability

Applications	Total Seeded Faults	Detected by BigTest	Detected by Sedge	Injected Fault Type 1	2	3	4	5	6	7
Income Aggregate	3	3	1	✓	NA	NA	✓	NA	NA	✓
Movie Rating	6	6	6	✓	✓	✓	✓	NA	✓	✓
Airport Layover	6	6	4	✓	✓	✓	✓	NA	✓	✓
Commute Type	6	6	4	NA	✓	✓	✓	✓	✓	✓
PigMix-L2	4	4	2	NA	✓	✓	NA	✓	✓	NA
Grade Analysis	4	4	3	NA	✓	✓	✓	NA	NA	✓
Word Count	2	2	0	NA	✓	NA	NA	NA	NA	✓

BigTest detects 2X more faults than Sedge because it models the internal semantics of UDFs with the specifications of dataflow operators.

RQ3: Test Size Reduction



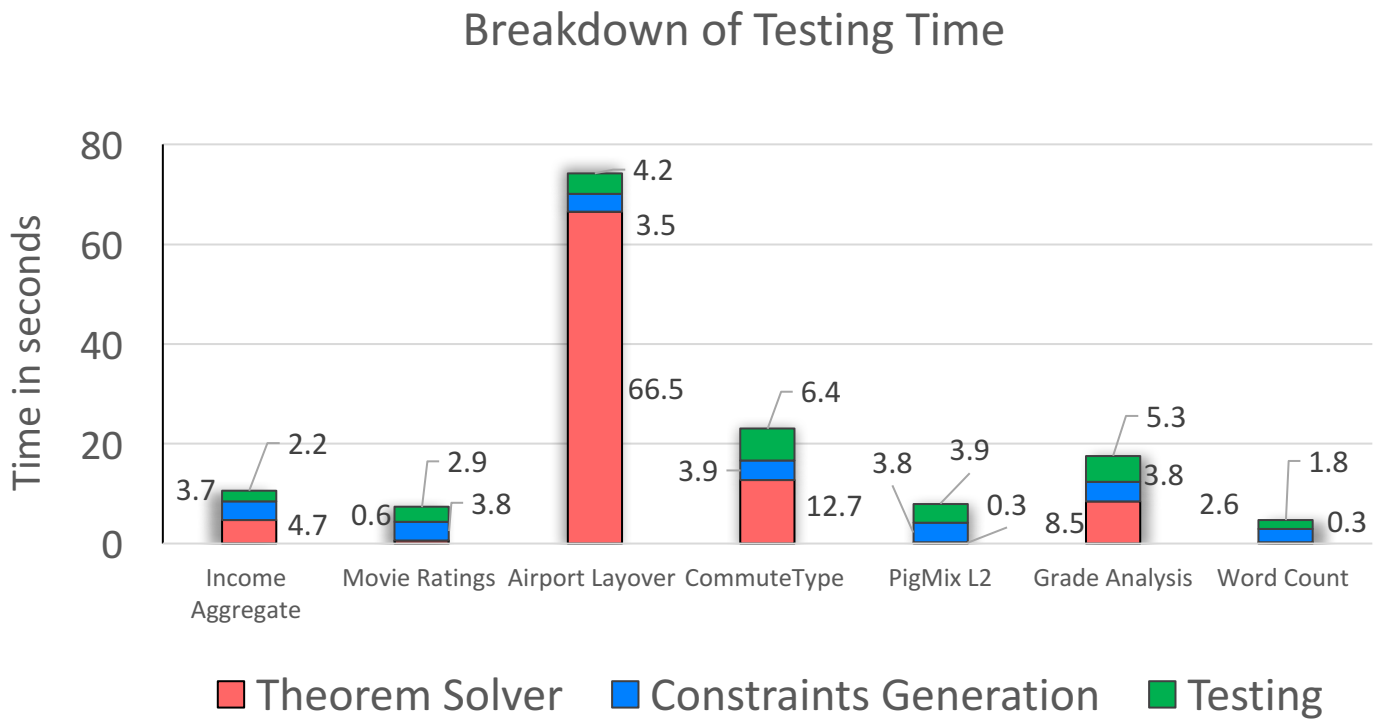
Compared to the entire dataset, BigTest achieves more JDU path coverage with 10^5X to 10^8X smaller test data, translating in to 194X testing speed up.

Summary

- Need SE tools for big data analytics applications
- BigTest provides ***exhaustive***, ***automatic***, and ***fast*** testing
- Contributions:
  1. Demonstrated the need to interpret UDFs
  2. Model strings, collections, and tuples
  3. Logical specifications for dataflow operators handling terminating and nonterminating cases
  4. Provide the first symbolic execution engine for Apache Spark/Scala
  5. Present a study of big data analytics bugs and the first bug benchmark

Publically available at:
https://github.com/maligulzar/BigTest

RQ3: Breakdown of BigTest's Testing Time



By running tests locally, BigTest improves the testing time (CPU seconds) by 194X, on average, compared to testing the entire dataset on 16-node cluster.

Inadequate Test Generation Tools for Big Data Analytics

Traditional Software Test Generation

```
def concat(append: boolean, a:
String, b: String ) {
   result: String = null;
   If (append)result = a + b;
   return
result.toLowerCase();
}
```

- Standalone application
- Symbolic Execution Compatible
- Well defined semantics
- Logical execution is similar to physical execution

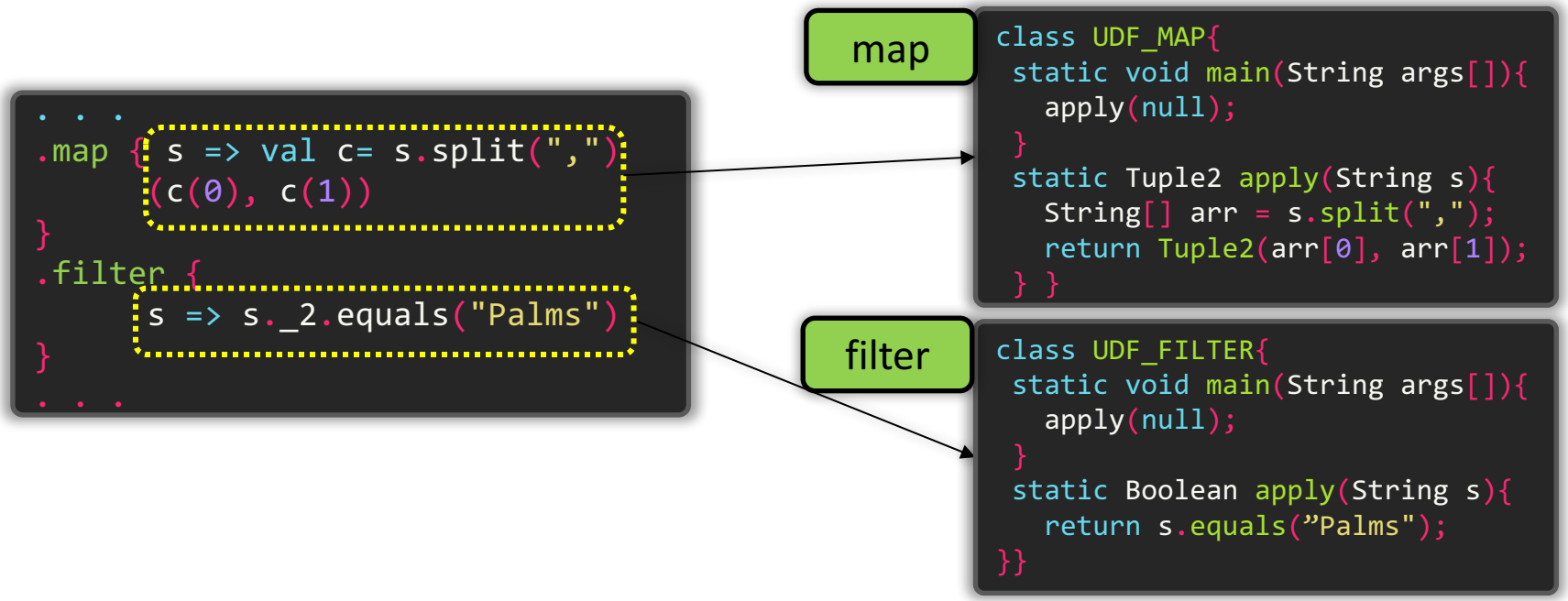
Big Data Analytics Test Generation

```
sc.textFile("hdfs")
.flatMap(s=> s.split(","))
.map(w =>(w,1))
.reducebyKey(_+_)
```

- Heavily depends on framework
- Non-existence Symbolic Execution for dataflow operators
- New operators with changing semantics
- Logical execution is different to physical execution

Program Decomposition

- **Challenge:** Due to the complexity of DISC frameworks' code, symbolic execution is infeasible on DISC applications.
- **Insight:** The individual UDFs of DISC application are relatively smaller (<100 LOC) making symbolic execution feasible**.**
- **Solution:** We decompose a DISC application using AST analysis into a set of individual UDFs and dataflow operators.



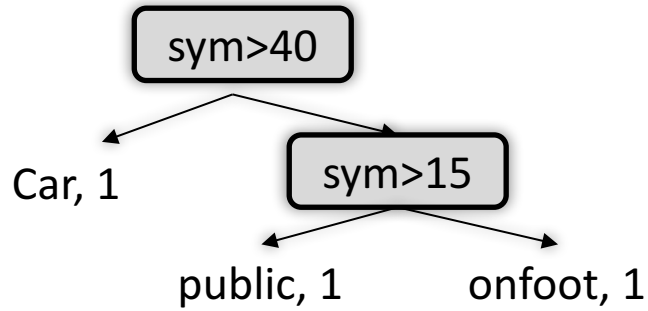
Symbolic Execution of UDFs

- **Challenges:** Strings, Collections, and Object are eminent in DISC applications but not fully support by symbolic execution tool *i.e* Java PathFinder.
- **Insight**: In DISC applications, most unbounded types are eventually bounded. We perform lazy SE on such types *e.g* `Split(“,”)` is unbounded Array but `Split(“,”)[1]` is bounded.
- **Solution:** Using JPF, we symbolically execute UDFs in isolation to generated path constraints and effects. Loops and Arrays are bounded by K=2.

map

From Step 1

```
class UDF{
 static void main(String args[]){
   apply(null);
 }
 static Tuple2 apply(Tuple3 s){
   if (s._2()._1() > 40)
     return Tuple2("car", 1);
   else if (s._2()._1() > 15)
     return Tuple2("public", 1);
   else
     return Tuple2("onfoot", 1);
 }
}
```



Path Constraints	Effect
sym > 40	Car , 1
40 ≥ sym>15	Public , 1
sym ≤ 15	onfoot , 1

Logical Specifications of Dataflow Operators

- **Challenges:** Dataflow operators in DISC applications are accompanied with 100Ks lines of framework code making symbolic execution infeasible.
- **Insight:** Dataflow operators have standard semantics but implemented differently for optimization purposes.
- **Solution:** Using these semantics, we abstract their implementation in logical specifications and used the specifications to tie together UDFs' symbolic trees.