Automated and Interactive Debugging of Big Data Analytics

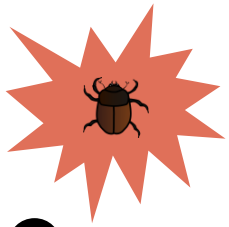
Muhammad Ali Gulzar
University of California, Los Angeles

Big Data Debugging in the Dark

1 Develop locally

2 Hope it works

3 Run in cloud



4 Bug!

5 Guesswork

Google

Map Reduce

hadoop

Spark

HIVE

ICSE ’16	• Interactive Debugging Primitives for Big Data Processing in Spark
SoCC ’17	• Automated Debugging in Data Intensive Scalable Computing Systems
Ongoing	• White-box Testing of Data Intensive Scalable Computing Applications with user defined functions

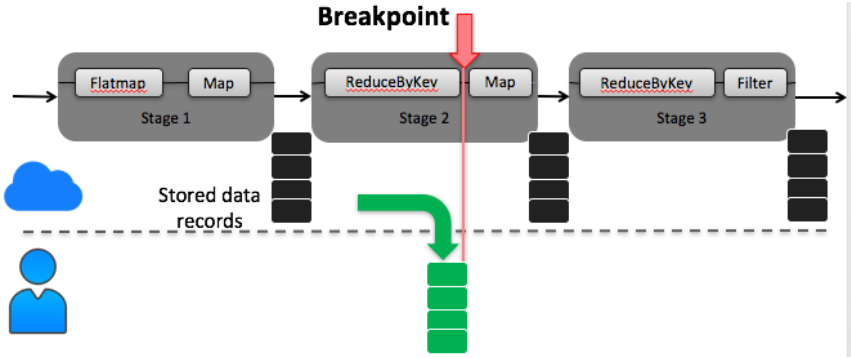
Why Traditional Interactive Debugging is Hard for Apache Spark?

Enabling interactive debugging requires us to **re-think the features of traditional debugger** such as GDB

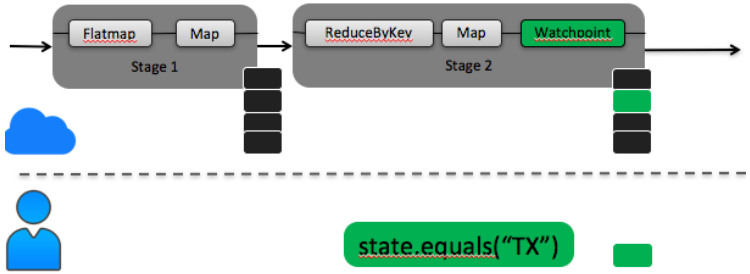
- Pausing the entire computation on the cloud could reduce throughput
- It is clearly infeasible for a user to inspect billion of records through a regular watchpoint
- Even launching remote JVM debuggers to individual worker nodes cannot scale for big data computing

Interactive DISC Debug Primitives [ICSE '16, FSE'16 Demo, SIGMOD'17 Demo]

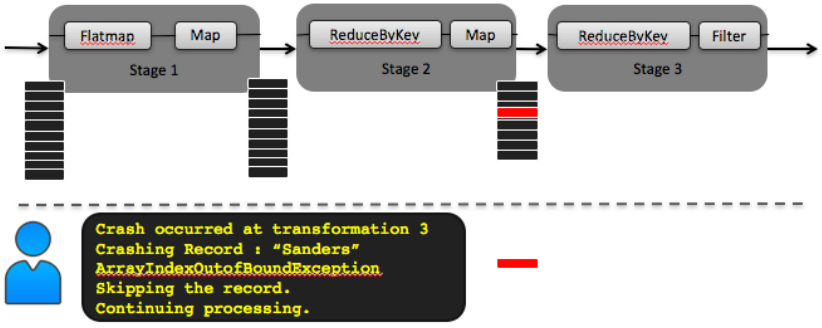
1. Simulated Breakpoint



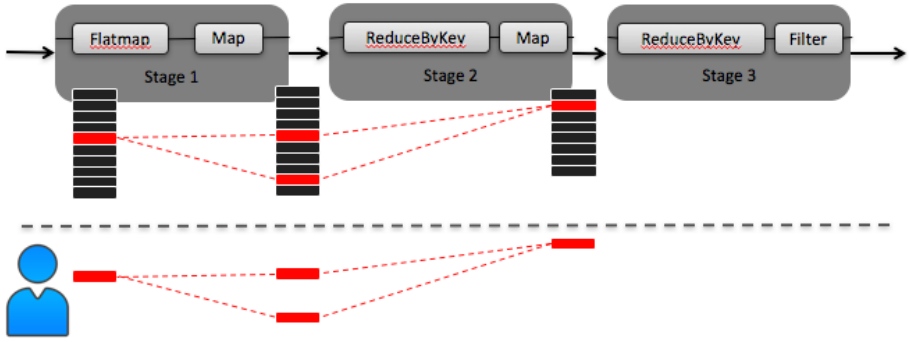
2. On Demand Guarded Watchpoint



3. Crash Culprit Identification



4. Backward and Forward Tracing



Our Insights for Interactive DISC Debugging

- We **do not pause program execution** but simulate a breakpoint through on-demand state regeneration
- We deliver **selected program states** to a user in a streaming processing fashion.
- We re-architect the underlying big data system runtime with native **in-memory data provenance support**

What is the performance overhead of debugging primitives?

Program	Dataset size (GB)	Max	Max w/o Latency Alert	Watchpoint	Crash Culprit	Tracing
WordCount	0.5 - 1000	2.5X	1.34X	1.09X	1.18X	1.22X
Grep	20 - 90	1.76X	1.07X	1.05X	1.04X	1.05X
PigMix-L1	1 - 200	1.38X	1.29X	1.03X	1.19X	1.24X

Max : All the features of BigDebug are enabled

BigDebug poses at most 2.5X overhead with the maximum instrumentation setting.

ICSE ’16	• Interactive Debugging Primitives for Big Data Processing in Spark
SoCC ’17	• Automated Debugging in Data Intensive Scalable Computing Systems
Ongoing	• White-box Testing of Data Intensive Scalable Computing Applications with user defined functions

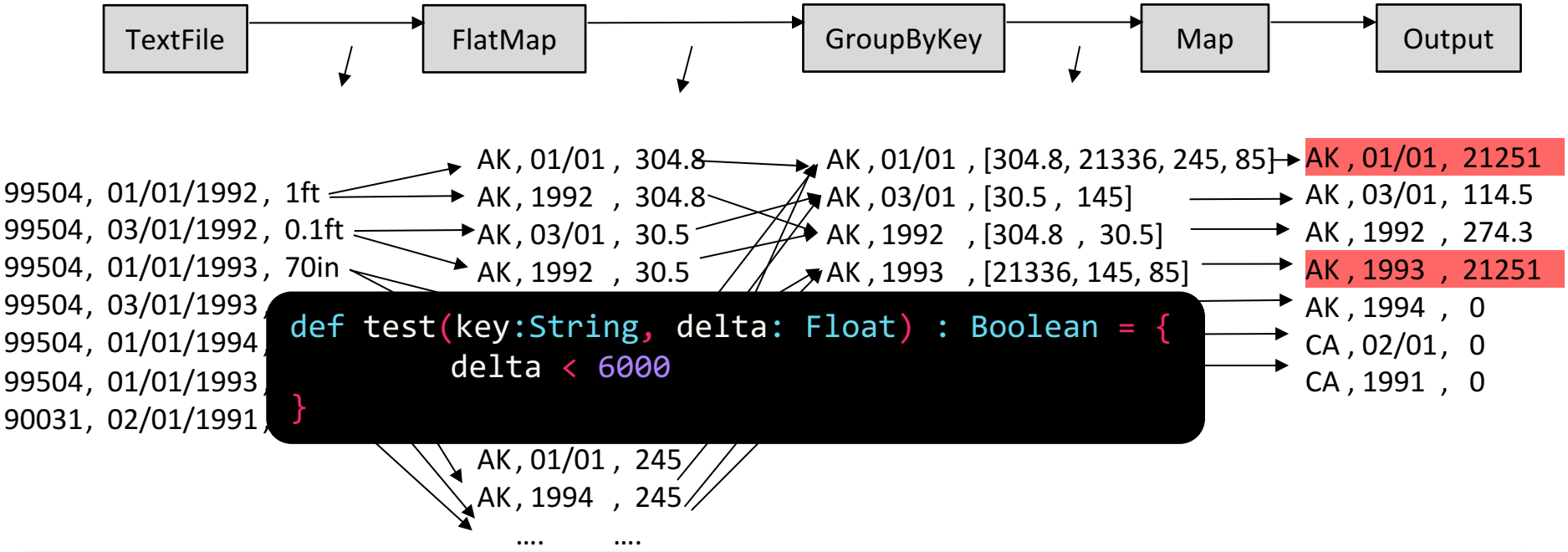
Motivating Example

- Alice writes a Spark program that identifies, **for each state** in the US, the **delta between the minimum and the maximum** snowfall reading for **each day of any year** and **for any particular year**.
- An input data record that measures 1 foot of snowfall on January 1st of Year 1992, in the 99504 zip code (Anchorage, AK) area, appears as

99504 , 01/01/1992 , 1ft

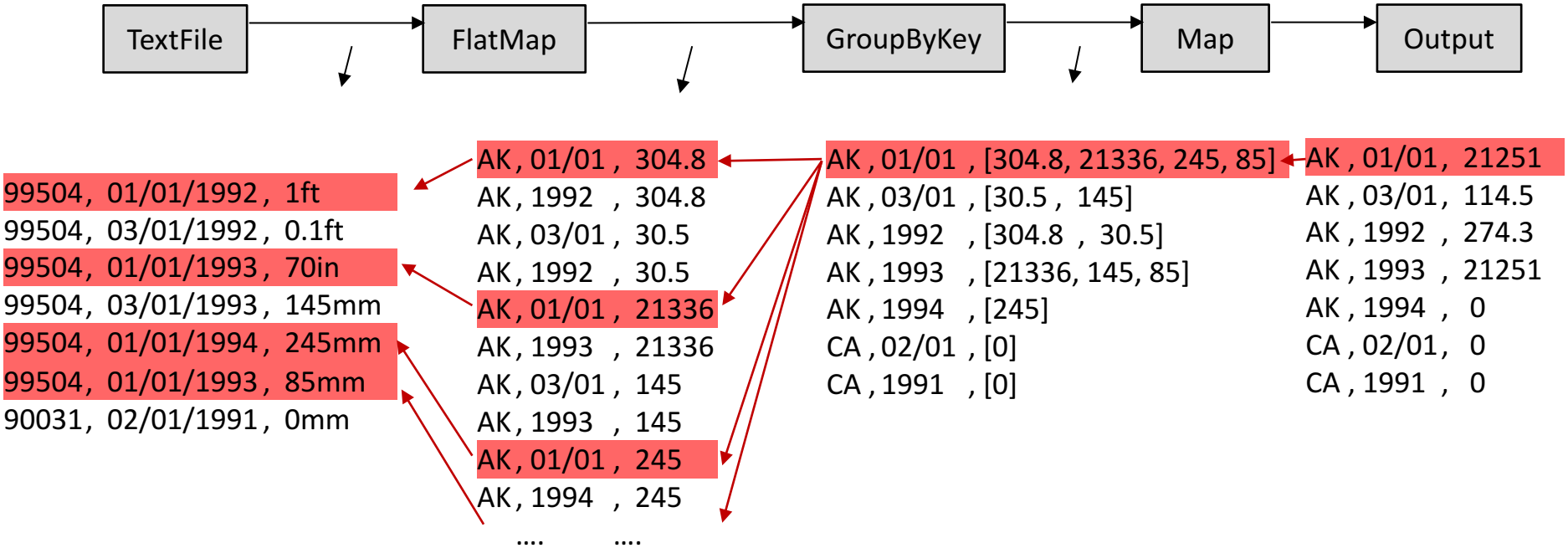
Problem Definition

- Using a test function, a user can specify incorrect results



Given a test function, the goal is to identify a minimum subset of the input that is able to reproduce the same test failure.

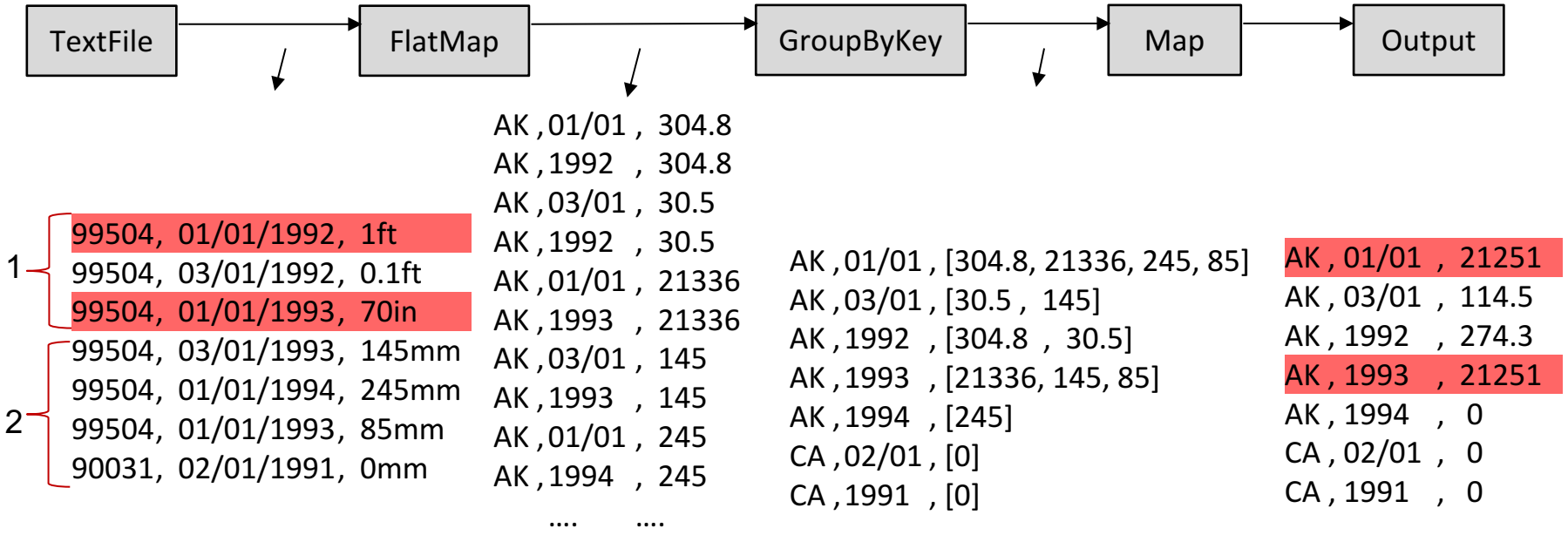
Existing Approach 1: Data Provenance for Spark [VLDB 2015]



It over-approximates the scope of failure-inducing inputs *i.e.* records in the faulty key-group are all marked as faulty

Existing Approach 2: Delta Debugging [Zeller 1999]

- Delta Debugging performs a systematic binary search-like procedure on the input dataset using a test oracle function

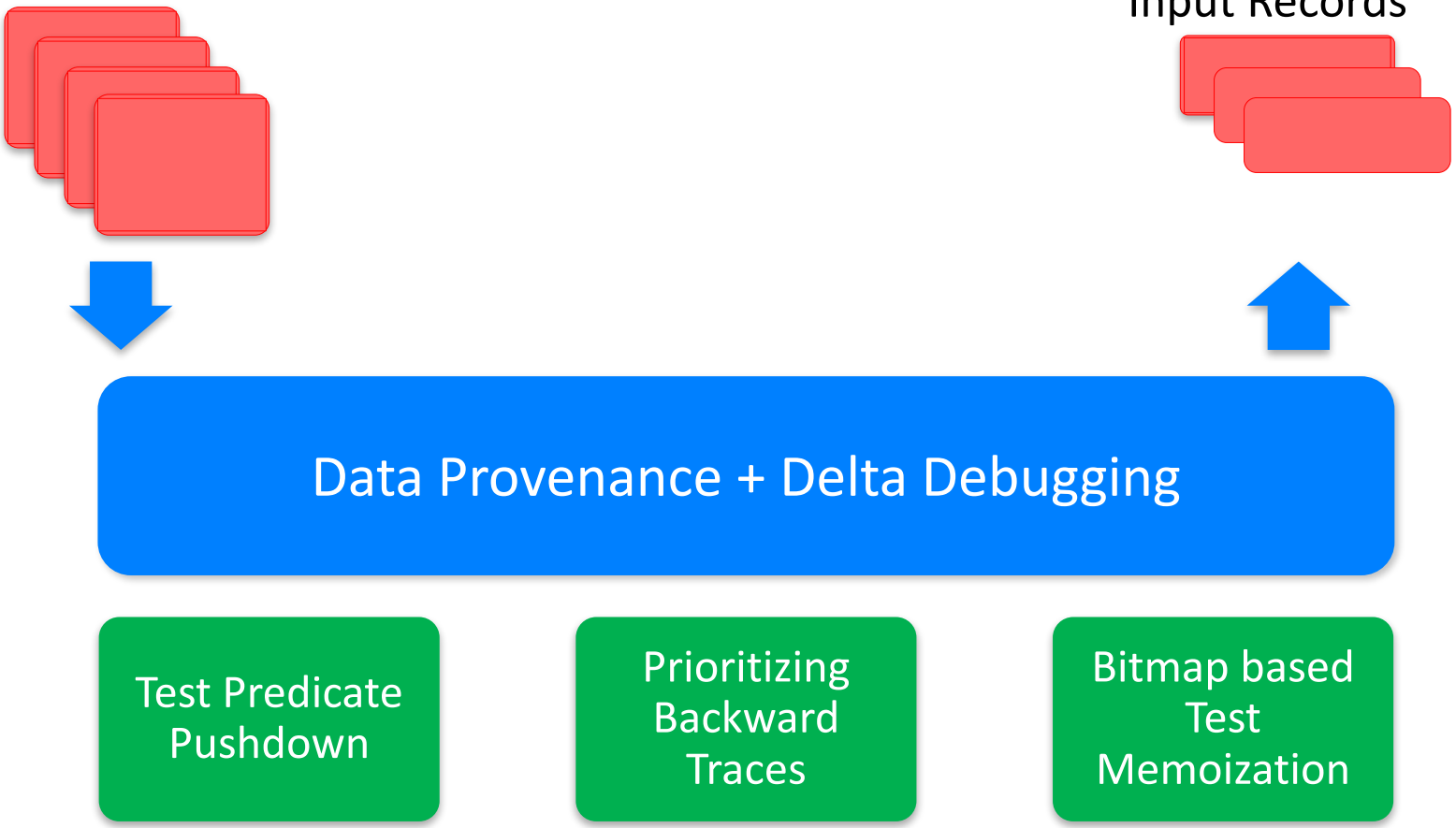


It does not prune input records known to be irrelevant because of the lack of semantic understanding of data-flow operators

Automated Debugging in DISC with BigSift [SoCC 2017]

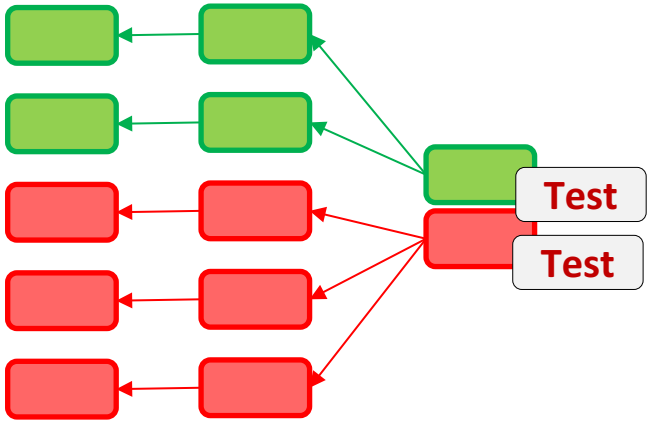
Input: A Spark Program, A Test Function

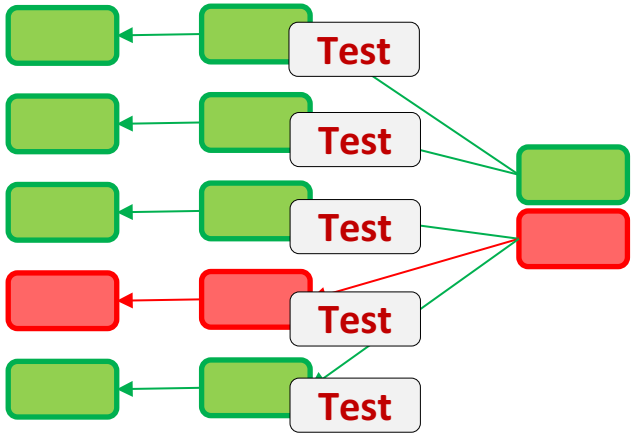
Output: Minimum Fault-Inducing Input Records



Optimization 1: Test Predicate Pushdown

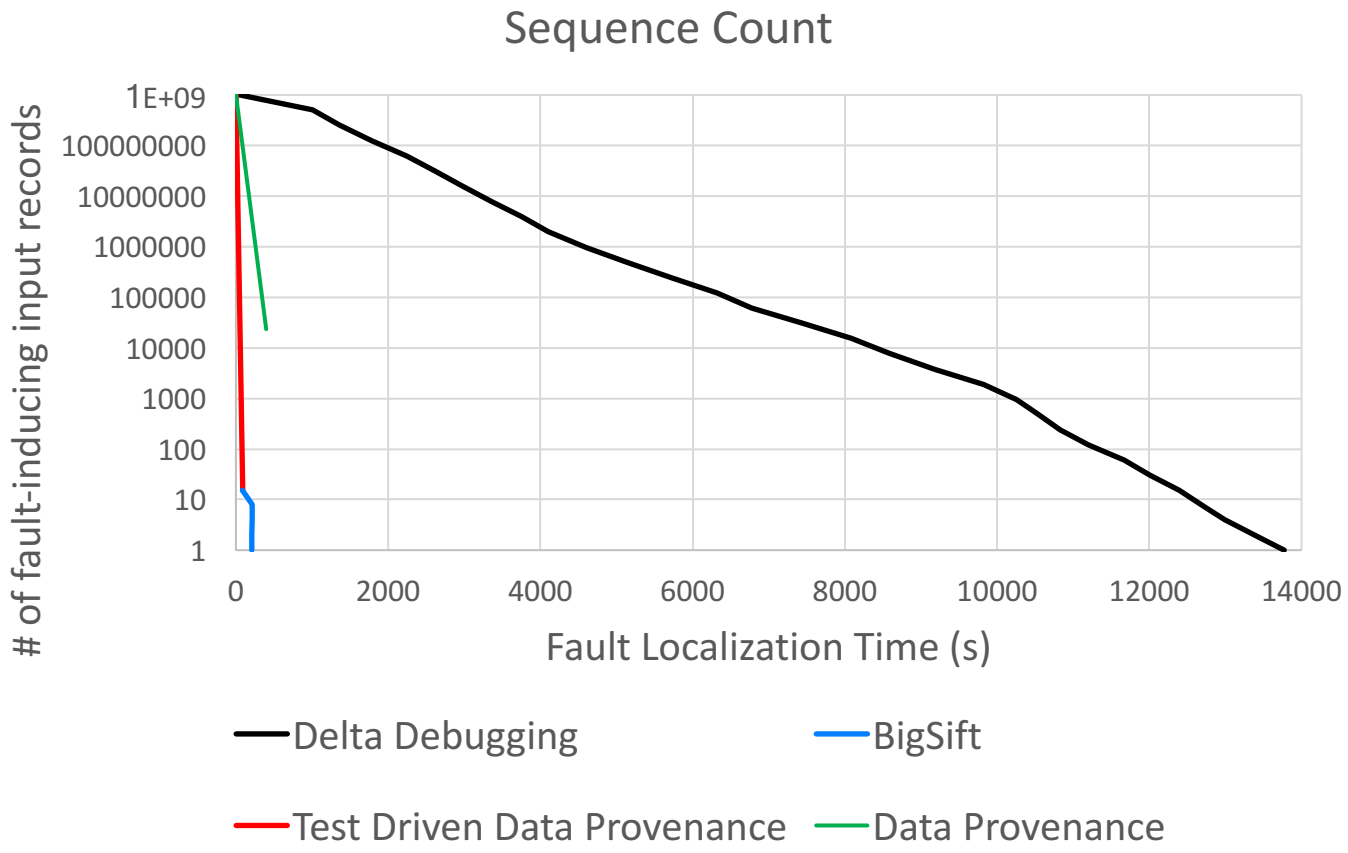
- **Observation:** During backward tracing, data provenance traces through all the partitions even though only a few partitions are faulty





If applicable, BigSift pushes down the test function to test the output of combiners in order to isolate the faulty partitions.

Debugging Time



On average, BigSift takes 62% less time to debug a single faulty output than the time taken for a single run on the entire data.

ICSE ’16	• Interactive Debugging Primitives for Big Data Processing in Spark
SoCC ’17	• Automated Debugging in Data Intensive Scalable Computing Systems
Ongoing	• White-box Testing of Data Intensive Scalable Computing Applications with user defined functions

Testing Challenges of Big Data Analytics

- How can we select a **minimal sample** from a complete dataset to perform efficient testing of DISC applications?
- How can we generate a data that **exercises all execution paths** in a DISC application to facilitate complete testing?
- Due to **dataflow operators** and complex **user defined functions** in DISC application, it is extremely hard to answer the two mentioned questions.

```
sc.textFile("hdfs://..")
      .flatMap(
      .map(
      .reduceByKey(
```

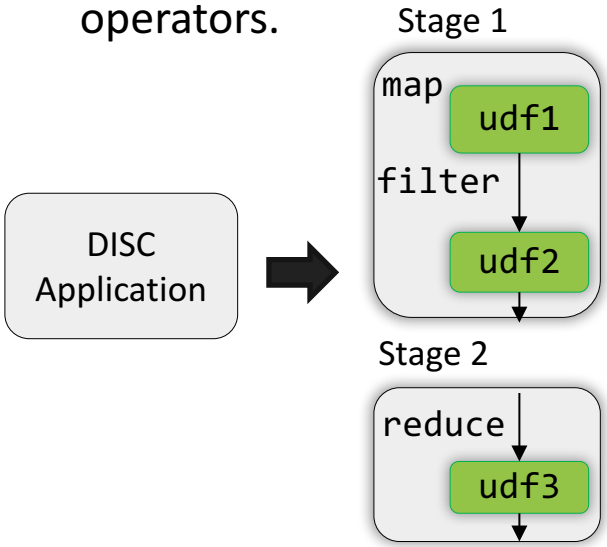
Dataflow Operators

```
s=>s.split(".") )
s => (s,1) )
(a,b) => a+b )
```

User defined functions

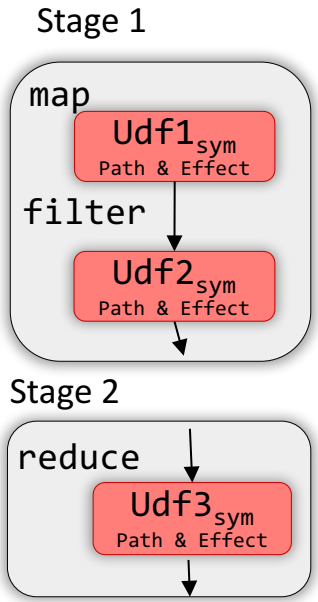
Ongoing work: White-box Testing of DISC Applications

1. A DISC application is decomposed into UDFs and dataflow operators.



2. Each complex UDF is symbolically executed in isolation with bounded path exploration.

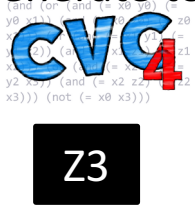
3. UDF Path constraints are integrated w.r.t the logical specifications of data flow operators



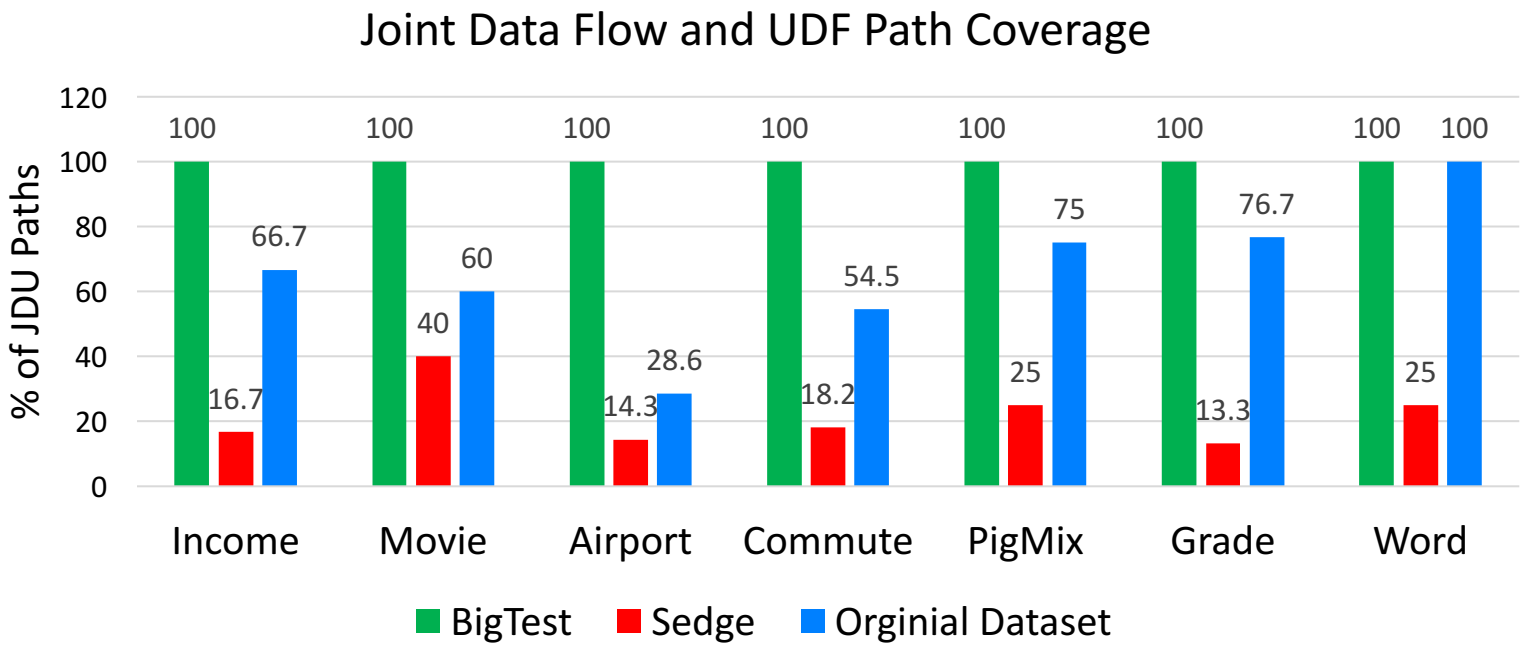
Path Constraint	Effects
X>5 & y= ..	z = x*…
X<3 & y>…	z = x/…
…	….

4. Path constraints are converted into SMT2 which is used to generate test data theorem solver

Path Constraint	Effects
X>5 & y= ..	z = x*…
X<3 & y>…	z = x/…



Test coverage from generated testing data



BigTest outperforms previously known technique and provides 100% JDU path coverage is almost every benchmark program

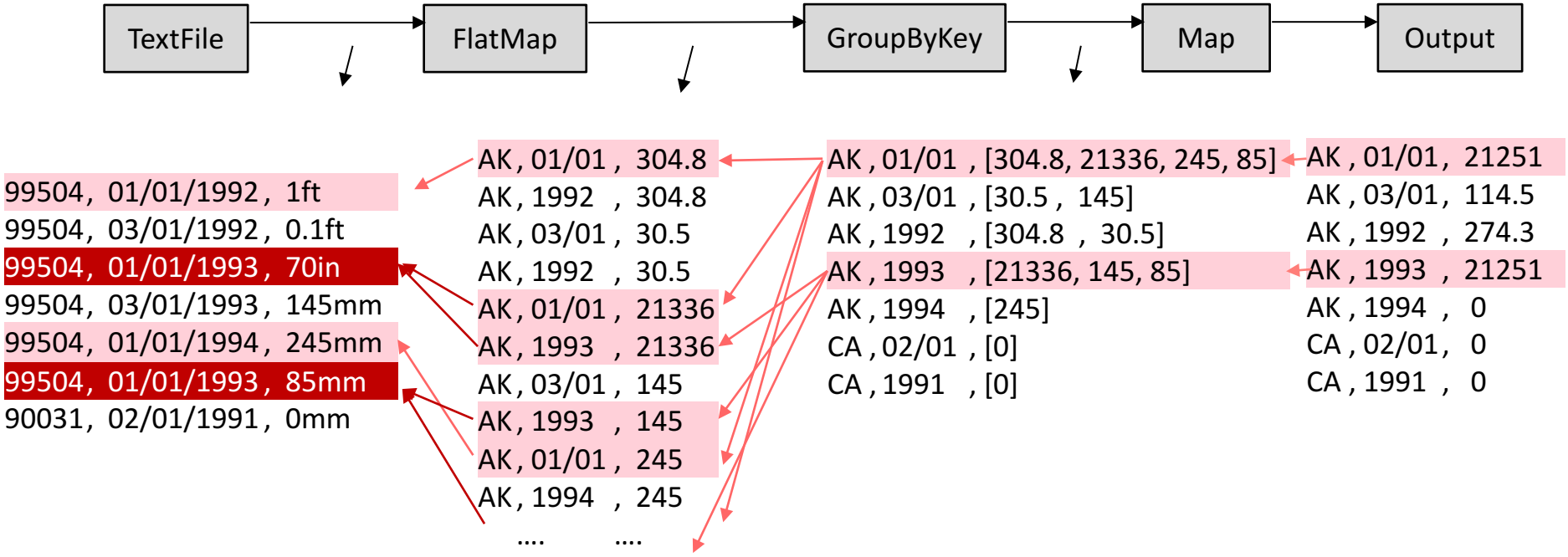
Conclusion

- By synthesizing insights from software engineering and database systems, we can design **scalable**, **interactive**, and **automated debugging** algorithms for big data analytics.

- Demo: Debugging Big Data Analytics in Spark with BigDebug
  https://www.youtube.com/watch?v=aZ91EyC5-Yc
- Demo: Automated Debugging of Big Data Analytics with BigSift
  https://www.youtube.com/watch?v=_HR3VJ2dPbE

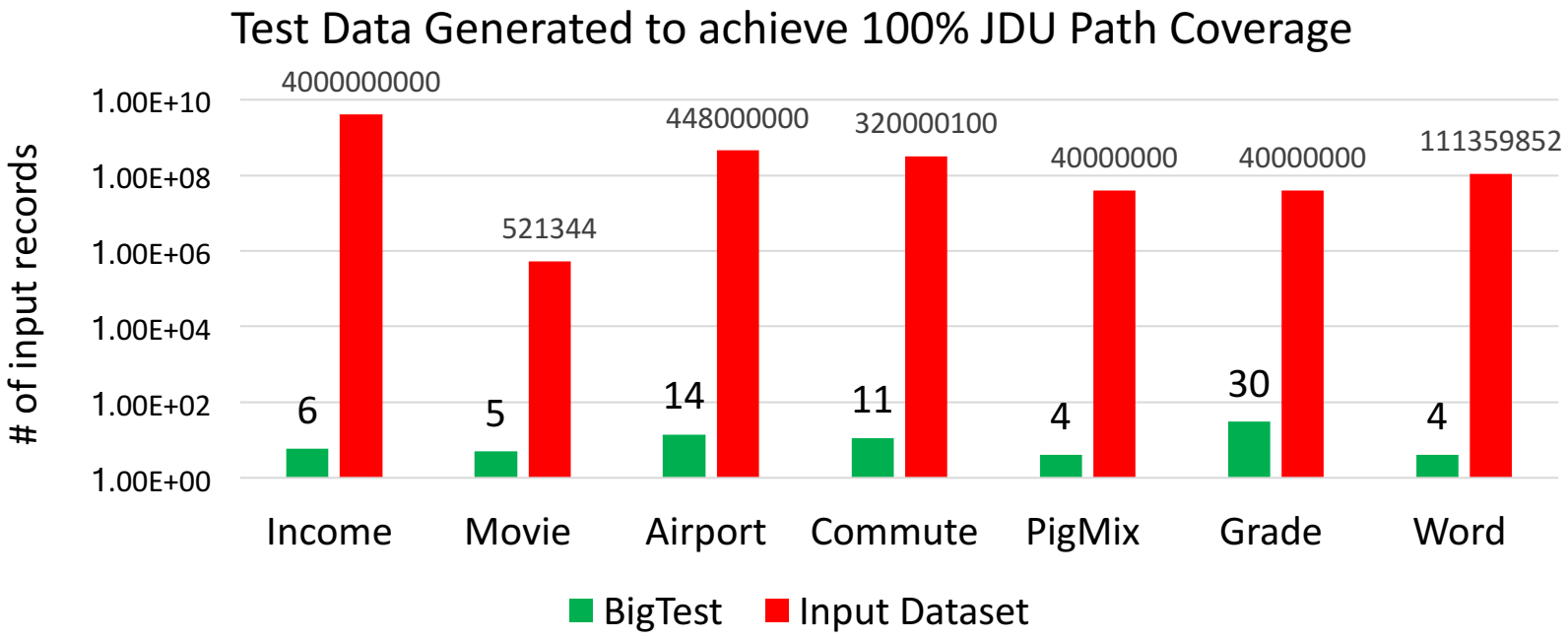
Optimization 2: Prioritizing Backward Traces

- **Observation:** The same faulty input record may contribute to multiple output records failing the test.



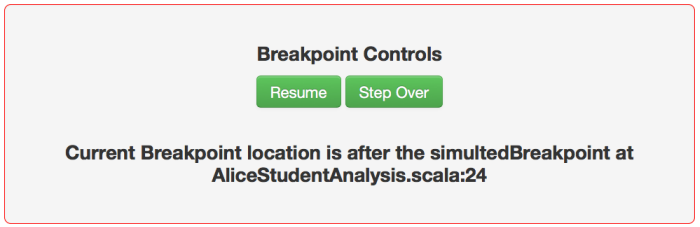
In case of multiple faulty outputs, BigSift overlaps two backward traces to minimize the scope of fault-inducing input records

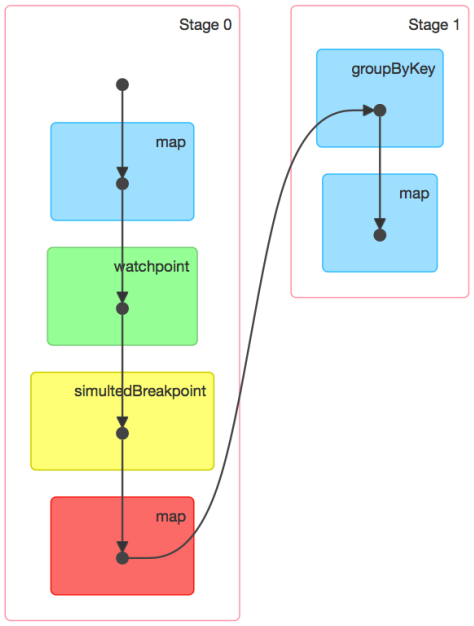
Test Data Size from BigTest



BigTest generates testing date of size that is several orders of magnitude (10^6-10^{10}) smaller than the original input dataset

BigDebug: Interactive Debugger [FSE 2016 Demo, SIGMOD 2017 Demo]



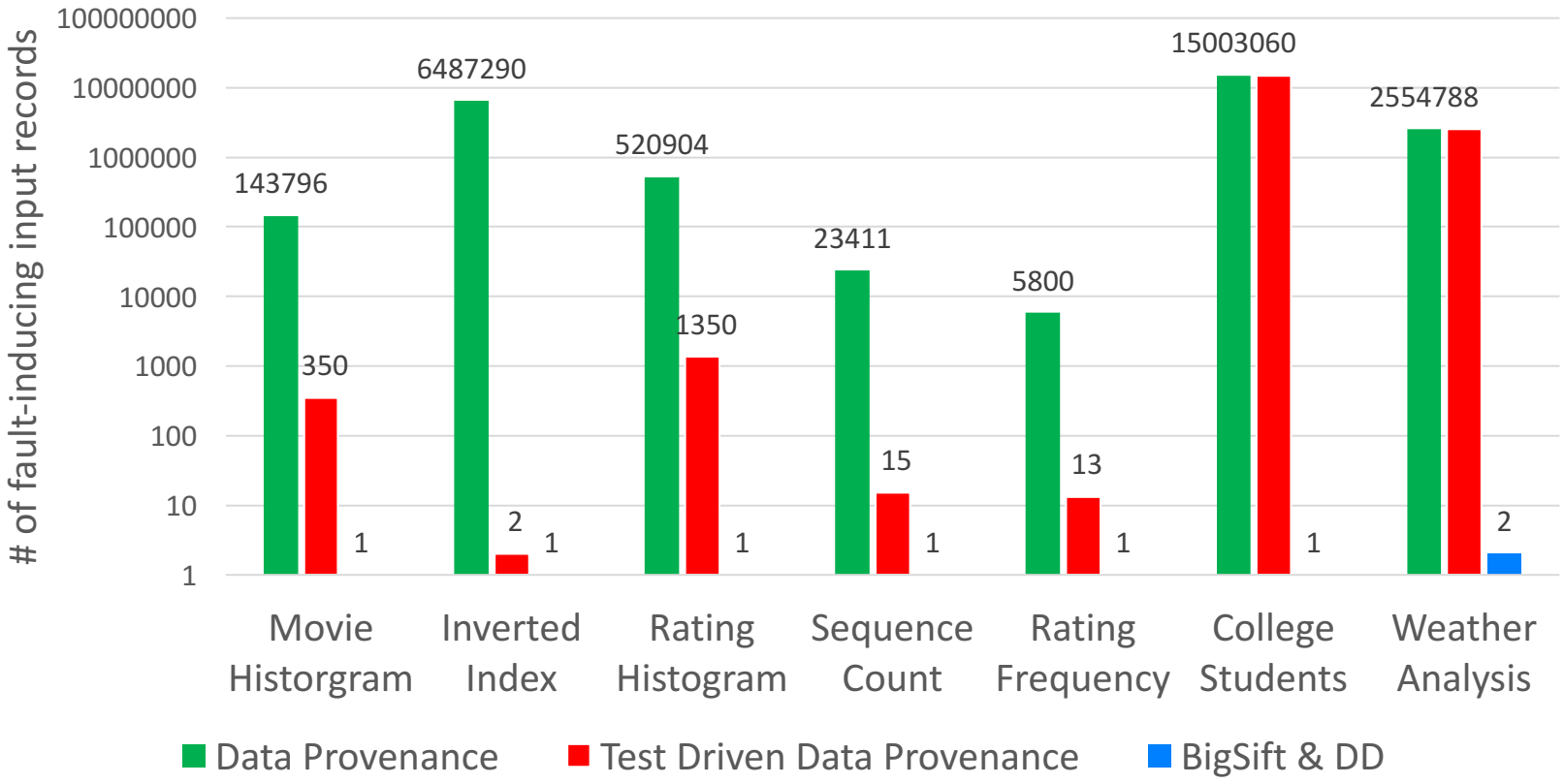


AliceStudentAnalysis.scala

```
10  object AliceStudentAnalysis {
11
12    val COLLEGEYEAR = List("Sophomore" , "Freshman" , "Junior", "Senior")
13    def main(args: Array[String]): Unit = {
14
15      //set up spark configuration
16      val sparkConf = new SparkConf()
17      val bdconf = new BigDebugConfiguration
18      bdconf.setFilePath("/home/ali/work/temp/git/dsbigdebug/spark-lineage/exa
19      //set up spark context
20      val ctx = new SparkContext(sparkConf)
21      ctx.setBigDebugConfiguration(bdconf)
22      //spark program starts here
23      val records = ctx.textFile("/home/ali/Desktop/myfile.txt", 1).
24      simultedBreakpoint(s=> !COLLEGEYEAR.contains(s.split(" ")(2)))
25>     val grade_age_pair = records.map(line => {
26        val list = line.split(" ")
27        (list(2), list(3).toInt)
28      })
29      val average_age_by_grade = grade_age_pair.groupByKey
30        .map(pair => {
31        val itr = pair._2.toIterator
32        var moving_average = 0
33        var num = 1
34        while (itr.hasNext) {
35          moving_average = moving_average + itr.next()
36          num = num + 1
37        }
38        (pair._1, moving_average/num)
39      })
40      val out = average_age_by_grade.collect()
41      out.foreach(println)
42    }
43  }
44
```

- BigDebug is publically available at https://sites.google.com/site/sparkbigdebug/

Fault Localizability over Data Provenance



BigSift leverages DD after DP to continue fault isolation, achieving several orders of magnitude 10^3 to 10^7 better precision.