Automated Debugging in Data-Intensive Scalable Computing

Muhammad Ali Gulzar[1], Matteo Interlandi[2], Xueyuan Han[3], Mingda Li[1], Tyson Condie[1], and Miryung Kim[1]

[1]*University of California, Los Angeles* [2]*Microsoft* [3]*Harvard University*

ABSTRACT

Developing Big Data Analytics workloads often involves trial and error debugging, due to the unclean nature of datasets or wrong assumptions made about data. When errors (*e.g.,* program crash, outlier results, etc.) arise, developers are often interested in identifying a subset of the input data that is able to reproduce the problem. BIGSIFT is a new faulty data localization approach that combines insights from automated fault isolation in software engineering and data provenance in database systems to find a minimum set of failure-inducing inputs. BIGSIFT redefines data provenance for the purpose of debugging using a test oracle function and implements several unique optimizations, specifically geared towards the iterative nature of automated debugging workloads. BIGSIFT improves the accuracy of fault localizability by several orders-of-magnitude ($\sim 10^3$ to $10^7\times$) compared to Titian data provenance, and improves performance by up to 66× compared to Delta Debugging, an automated fault-isolation technique. For each faulty output, BIGSIFT is able to localize fault-inducing data within 62% of the original job running time.

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; **Software testing and debugging**; • **Information systems →** *Data cleaning*; *Data provenance*;

KEYWORDS

Automated debugging, fault localization, data provenance, data-intensive scalable computing (DISC), big data, and data cleaning

ACM Reference Format:
Muhammad Ali Gulzar[1], Matteo Interlandi[2], Xueyuan Han[3], Mingda Li[1], Tyson Condie[1], and Miryung Kim[1]. 2017. Automated Debugging in Data-Intensive Scalable Computing. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 15 pages.
https://doi.org/10.1145/3127479.3131624

The work was conducted while all authors were at UCLA.

1 INTRODUCTION

Data-Intensive Scalable Computing (DISC) systems such as Google's MapReduce [18], Apache Spark [49], and Apache Hadoop [1] draw valuable insights from massive data sets to help make business decisions and scientific discoveries. Similar to other software development platforms, developers often deal with program errors and incorrect inputs *e.g.,* unclean data or making the wrong assumptions about the data. Furthermore, DISC systems provide increased expressiveness through user-defined functions, which consequently increases the complexity of debugging. It is therefore crucial to equip these developers with toolkits that can better pinpoint the root cause of an error. Otherwise, they might be forced to resort to an extremely lengthy and expensive process of manual trial and error debugging.

When a failure or incorrect result is generated (*e.g.,* outlier), the programmer may want to pinpoint the root cause by investigating the relevant subset of failure-inducing input records. One possible approach is to use *Data Provenance* (DP) to trace back to the input records responsible for inducing the error [6, 7, 17, 24, 26, 36] or generate data summaries of tracing queries [5, 35]. Another approach is to perform a systematic search on the input dataset using a test oracle function to isolate a minimum set of fault-inducing input records, which is a technique called *Delta Debugging* (DD) [51]. However, these approaches are not suitable for debugging DISC workloads for several reasons. First, DD does not consider the semantics of data-flow operators such as `join` and `group-by`, and thus cannot prune input records known to be irrelevant. Second, DD's search strategy is iterative: it re-runs the same program using different subsets of the input records, which is prohibitively expensive for tens or even hundreds of iterations on large datasets. Third, DP over-approximates the scope of failure-inducing inputs by considering that all intermediate inputs mapping to the same key contribute to the erroneous output.

To overcome these limitations, we present BIGSIFT, a new approach that brings automated debugging to a reality in DISC environments. Given a test function, BIGSIFT automatically finds a minimum set of fault-inducing input records responsible for a faulty output. We re-define data provenance [28] for the purpose of debugging by leveraging the semantics of data transformation operators. BIGSIFT then prunes out input records irrelevant to the given faulty output records, significantly reducing the initial scope of failure-inducing records before applying DD. We also implement a set of optimization and prioritization techniques that uniquely benefit the iterative nature of DD workloads. For example, we overlap the backward trace of multiple faults, based on the insight that a single culprit record may propagate to multiple output records. We implement bitmap based memoization and adaptive local job scheduling to speed up the debugging time. Our current implementation targets Apache Spark [49], a state of the art DISC system, but it can be generalized to any data processing system that supports data provenance.

In our evaluation, we compare BIGSIFT with baseline DD in terms of response time, and DP in terms of minimizing failure-inducing input records. We construct our own debugging benchmarks by porting the PUMA benchmark to Spark [4]. In addition to seeding fault-inducing records in the input data, we inject programming errors in code. This is to demonstrate BIGSIFT's capability to find faulty data records, where the notion of faulty data changes depending on coding errors. Such faults cannot be found by data cleaning techniques that do not consider interaction between input data and code.

In comparison to using DP alone, BIGSIFT finds a more concise subset of fault-inducing input records, improving its fault localization capability by several orders of magnitude. In most subject programs, data provenance stops at identifying failure inducing records at the size of up to $\sim10^3$ to 10^7 records, which is still infeasible for developers to manually sift through. In comparison to using DD alone, BIGSIFT reduces the fault localization time (as much as 66×) by pruning out input records that are not relevant to faulty outputs. Further, our trace overlapping heuristic decreases the total debugging time by 14%, and our test memoization optimization provides up to 26% decrease in debugging time. Indeed, the total debugging time taken by BIGSIFT is often 62% less than the original job running time per single faulty output. In software engineering literature, the debugging time is generally much longer than the original running time [11, 16, 51].

The rest of the paper is organized as follows. Section 2 provides a brief introduction to Apache Spark. Section 3 describes a motivating example. Section 4 describes the design and implementation of BIGSIFT. Section 5 describes evaluation settings and the corresponding results. Section 6 discusses related work.

2 BACKGROUND: APACHE SPARK

Apache Spark [2] is a widely used large scale data processing platform that achieves orders-of-magnitude better performance than Hadoop MapReduce [1] for iterative workloads. BIGSIFT targets Spark because of its wide adoption and support for interactive ad-hoc analytics. The Spark programming model can be viewed as an extension to the Map Reduce model with direct support for traditional relational algebra operators (*e.g.,* group-by, join, filter) and iterations. Spark programmers leverage Resilient Distributed Datasets (RDDs) to apply a series of transformations to a collection of data records (or tuples) stored in a distributed fashion *e.g.,* in HDFS [43]. Calling a transformation on an RDD produces a *new* RDD that represents the result of applying the given transformation to the input RDD. Transformations are lazily evaluated. 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) of *stages*, where each stage contains some sub-series of transformations until a *shuffle step* is required (*i.e.,* data must be re-partitioned). The Spark scheduler is responsible for executing each stage in a topological order, with *tasks* performing the work of a stage on input partitions. Each stage is fully executed before downstream dependent stages are scheduled. The action result values are collected from the final output stage and returned to the user. Apache Spark allows developers to cache results. Additionally, by default Spark materializes intermediate results for fault tolerance. In particular, at each shuffle step, all the intermediate results of a current job are materialized before proceeding to the next stage.

```
1   val log = "s3n://xcr:wJY@ws/logs/weather.log"
2   val split = sc.textFile(log).flatMap{s =>
3    val tokens = s.split(",")
4    // finds the state for a zipcode
5    var state = zipToState(tokens(0))
6    var date = tokens(1)
7    // gets snow value and converts it into millimeter
8    val snow = convertToMm(tokens(2))
9    //gets year
10   val year = date.substring(date.lastIndexOf("/"))
11   // gets month / date
12   val monthdate= date.substring(0,date.lastIndexOf("/")-1)
13     List[((String , String) , Float)](
14         ((state , monthdate) , snow) ,
15         ((state , year) , snow)
16       )
17  }
18  val deltaSnow = split.groupByKey().map{ s =>
19   val delta = s._2.max - s._2.min
20   (s._1 , delta)
21  }
22  deltaSnow.saveAsTextFile("hdfs://s3-92:9010/output/")
23  def convertToMm(s: String): Float = {
24   val unit = s.substring(s.length - 2)
25   val v = s.substring(0, s.length - 2).toFloat
26   unit match {
27    case "mm" => return v
28    case _ => return v * 304.8f
29    }
30  }
```

Figure 1: Alice's 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. Measurements can be either in millimeters or in feet. The conversion function is described at line 23.

3 MOTIVATING EXAMPLE

This section discusses a motivating example to elucidate the challenges of debugging DISC system workloads and the limitations of DD and DP approaches in addressing this challenge.

Alice writes a Spark program to process a large dataset that contains weather telemetry data of the U.S. over several years. She wants to compute the delta between the minimum and the maximum snowfall measurement in each state for (1) each day of any year and (2) for each year. Data records are in CSV format: for example, the following sample record indicates that on January 1st of Year 1992, in the 99504 zip code (Anchorage, AK) area, there was 1 foot of snowfall: `99504 , 01/01/1992 , 1ft` .

To analyze the data, Alice develops the Spark program shown in Figure 1. She starts projecting each base record into two records (lines 3-17); the first representing the state, the date (`mm/dd`), and its snowfall measurement, and the second representing the state, the year (`yyyy`), and its snowfall measurement. She normalizes the snowfall measurements using the function `convertToMm` (described at line 23), which converts any units of feet to millimeters, based on an assumption she makes about the data. She also uses a function `zipToState` (line 5) to find the name of the state where an input zip code resides.

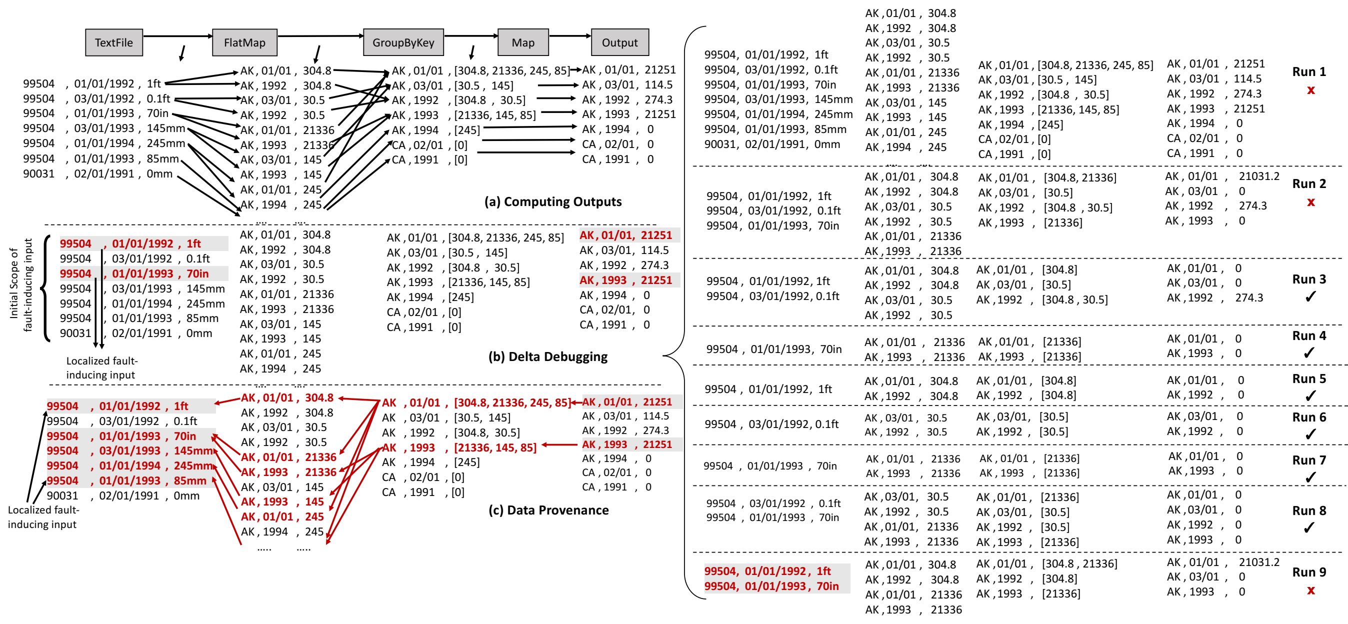


Figure 2: (a) shows how intermediate and final results are constructed using the transformations in Alice's program. In (b), delta debugging considers the initial scope of the fault-inducing input to span a complete dataset but it eventually finds the precise fault-inducing input. In (c), data provenance over-approximates the set of fault inducing input records because of the `groupByKey` operation in the initial program.

Next, she groups the key value pairs using a `groupByKey` operator in line 18, yielding records that are grouped in two ways (a) by state and day and (b) by state and year. At lines 18-21, Alice finds the delta between the maximum and the minimum snowfall measurements for each group and saves the final results to an HDFS directory. A snippet of the execution is shown in Figure 2(a), where, on January 1st, the snowfall level delta in Alaska (AK) is `21251` millimeters, and in year `1992`, the snowfall level delta in Alaska is `274.3` millimeters.

```
def test(key:String, delta: Float) : Boolean = {
  delta < 6000
}
```

Figure 3: Test function checking the validity of each output record—all snowfall deltas greater than 6000 millimeter are suspicious or incorrect.

Suppose that Alice writes a test function to check the validity of her output records, as seen in Figure 3. In her test function, she assumes that any delta snowfall level greater than 6000 millimeters (6 meters) is extremely suspicious, such as the delta snowfall of `21251`. However, once such outlier snowfall levels are identified, it is challenging for Alice to derive (by inspection) the precise set of input records leading to such faulty outputs, because the program involves computing both *min* and *max* over a unit conversion, making it hard to write a data cleaning filter upfront, as snowfall levels could vary greatly.

The goal of BIGSIFT is to identify the precise input records leading to each faulty output record. In this case, the faulty output records are caused by an error in the unit conversion code, because the developer could not anticipate that the snowfall measurement could be reported in the unit of *inches* and the default case converts the unit in feet to millimeters (line 28 in Figure 1). Therefore, the snowfall record `99504 , 01/01/1993 , 70in` is interpreted in the unit of *feet*, leading to an extremely high level of snowfall, like `21366` mm, after the conversion. In this case, BIGSIFT finds the minimum set of failure-inducing records: `99504 , 01/01/1992 , 1ft` and `99504 , 01/01/1993 , 70in`, from which the unit measurements can be inspected, prompting a correction in the `convertToMm` function.

Limitations of Delta Debugging. *Delta Debugging (DD)* addresses the problem of isolating failure-inducing inputs by repetitively running a program with different sub-configurations of input. DD splits the original input into two halves using a binary search-like strategy and re-runs them. If one of the two halves fails, DD recursively applies the same procedure for only that failure-inducing input configuration. On the other hand, if both halves pass, DD tries different sub-configurations by mixing fine-grained sub-configurations with larger sub-configurations (computed as the complement from the current configuration). Under the assumption that a failure is *monotone*—where C is a super set of all input configurations, if a larger configuration c is successful, then any of its smaller sub-configurations c' does not fail, i.e., $\forall c \subset C\ (\ test(c) = ✓ \rightarrow \forall c' \subset c\ (test(c') \neq ✗))$, DD returns a minimal failure-inducing configuration. The minimal failure-inducing configuration c_x means that removing any subset from c_x no longer fails: $\forall \delta_i \subset c_x, test(c_x - \{\delta_i\}) \neq ✗$.

One limitation of delta debugging is that it is a black box procedure that does not consider the semantics of underlying data flow operators. In our running example, since the faulty output

`AK, 01/01, 21251` is over state `AK` and date `01/01` only, we can easily conclude that the scope of failure-inducing input records should be limited to the records with date `01/01` and a zip code that exists in Alaska. However, delta debugging considers the entire input dataset (**Run 1**) as the initial scope of potential fault-inducing input, as it does not account for the semantics of the used data flow operators and keys.

Limitation of Data Provenance. *Data provenance (DP)* is a well-known technique in the database community for understanding the relationship between the input (or intermediate) records and the output records [6, 7, 17, 24]. For example, Titian is a data provenance tool for Apache Spark that allows users to perform forward and backward tracing of specific data records to understand the generation and consumption of data records [28]. If a user requests backward tracing for an output record, Titian identifies all the input records responsible for generating the output record from a series of transformations. As such, DP could identify a subset of the dataset containing the failure-inducing inputs by backward tracing from the faulty output(s).

Assume now that Alice uses Titian to perform backward tracing of the faulty output record. Titian creates correspondences from the intermediate records `AK , 01/01 , 304.8` `AK , 01/01 , 21336` `AK , 01/01 , 245` `AK , 01/01 , 85` to a single output `AK , 01/01 , 21251`, as seen in Figure 2(c). Though only two of the input records (the minimum and the maximum value) contributed towards the final output across `groupByKey` and `map`, DP over-approximates, by a significant amount, the scope of fault-inducing records by simply assuming that all input records in the list to `groupByKey` contributed to the output record.

4 APPROACH

BIGSIFT implements a unique combination of data provenance (DP) and delta debugging (DD) to offer a toolkit for efficiently debugging DISC system workloads. It accepts a DISC program, an input dataset, and a user-defined test function that distinguishes the faulty outputs from the correct ones. It then executes the DISC program, and uses the test function to identify faulty output records. The debugging process is performed in three phases.

Phase 1 applies *test driven data provenance* (TP) to remove input records that are not relevant for identifying the fault(s) in the initial scope of fault localization. BIGSIFT re-defines the notion of data provenance by taking insights from *predicate pushdown* [44]. By pushing down a *test oracle function* from the final stage to an earlier stage, BIGSIFT tests partial results instead of final results, dramatically reducing the scope of fault-inducing inputs. In Phase 2, BIGSIFT prioritizes the backward traces by implementing *trace overlapping*, based on the insight that faulty outputs are rarely independent *i.e.,* the same input record may propagate to multiple output records through operators such as `flatMap` or `join`. BIGSIFT also prioritizes the smallest backward traces first to explain as many faulty output records as possible within a time limit. In Phase 3, BIGSIFT performs *optimized delta debugging* while leveraging *bitmap based memoization* to reuse the test results of previously tried sub-configurations, when possible. Eventually, BIGSIFT outputs the smallest subset of input records responsible for the test failure of each faulty output. The debugging process of BIGSIFT is illustrated in Algorithm 1.

Algorithm 1 BIGSIFT's algorithm

```
    $local\_threshold$: an input size threshold on jobs for local computation
    $test(c)$ runs the program on configuration c and checks whether it fails the test, $test(c_✗)$ =✗ and $test(\emptyset)$ =✓
    $testCombiners(t, I)$ filters the partial result that fails test t
    $faults$: a minimum set of fault inducing input records
    $split(c, n)$ splits the input c into n configurations
 1: if $combinersForLastOperation$ then                      ▷ Phase I: Test Pushdown
 2:     $faulty\_output = testCombiners(test, input)$
 3: else
 4:     $faulty\_output = testOutput(test, input)$
 5: $C_L \leftarrow getLineage(faulty\_output)$                ▷ Phase I: Data Provenance
 6: $C_L = SmallestJobFirst(C_L)$                            ▷ Phase II: Smallest Job First
 7: while $!C_L.isEmpty()$ do
 8:     if $|C_L| > 1$ then
 9:         $C_L, c_{INT} = overlap(C_L)$                    ▷ Phase II: Trace Overlapping
10:         $faults.push(ddmin_2(c_{INT}, 2))$
11:         $faults.push(ddmin_2(C_L.pop(), 2))$
12:         $faults.push(ddmin_2(C_L.pop(), 2))$
13:     else
14:         $faults.push(ddmin_2(C_L.pop(), 2))$
15: return $faults$
16: function $ddmin_2(c_✗, n)$                               ▷ Phase III: Delta Debugging
17:     $C \leftarrow split(c_✗, n)$
18:     $(\Delta_i, testResult) = submitJob(C)$
19:     if $testResult == ✗$ then
20:         return $ddmin_2(\Delta_i, 2)$
21:     for $\Delta_i \in C$ do
22:         $C[i] = c_✗ - \Delta_i$
23:     $(\Delta_i, testResult) = submitJob(C)$
24:     if $testResult == ✗$ then
25:         return $ddmin_2(\Delta_i, max(n-1, 2))$
26:     if $n < |c_✗|$ then
27:         return $ddmin_2(c_✗, min(|c_✗|, 2n))$
28:     else
29:         return $c_✗$
30: function $submitJob(C)$
31:     $testResult = ✓$
32:     for $\Delta_i \in C$ do
33:         if $isTestMemoized(\Delta_i)$ then
34:             $testResult = getTestResult(\Delta_i)$
35:         else                                             ▷ Phase III: Adaptive Scheduling
36:             if $|\Delta_i| > local\_threshold$ then
37:                 $(\Delta_i, testResult) = runOnSpark(test, \Delta_i)$
38:             else
39:                 $(\Delta_i, testResult) = runOnLocal(test, \Delta_i)$
40:         $memoize(\Delta_i, testResult)$                  ▷ Phase III: Test Memoization
41:         if $testResult == ✗$ then
42:             return $(\Delta_i, testResult)$
43:     return $(\emptyset, testResult)$
44: function $overlap(C_L)$
45:     $c_{INT} \leftarrow C_L(0) \cap C_L(1)$
46:     if $test(C_{INT}) == ✗$ then
47:         $C_L(0) = C_L(0) - c_{INT}$
48:         $C_L(1) = C_L(1) - c_{INT}$
49:         return $(C_L, c_{INT})$
50:     return $(C_L, \emptyset)$
```

4.1 Phase I: Test Driven Data Provenance

In test-oracle based debugging such as DD, faulty outputs are distinguished from correct ones using a user-defined test function. Therefore, we could invoke a backward tracing query on each faulty output using a data provenance technique Titian to reduce the initial scope of fault-inducing inputs [28]. We describe how to use basic data provenance to identify an initial scope, and then how BIGSIFT extends it for test-oracle based debugging.

Data Provenance. When a Spark job is submitted, its workflow is generated in the form of a DAG. Titian takes in that DAG and inserts tracing agents in the workflow. These tracing agents modify the data records by attaching an identifier to each individual record. At every stage boundary, these ids are collected and added to an agent table

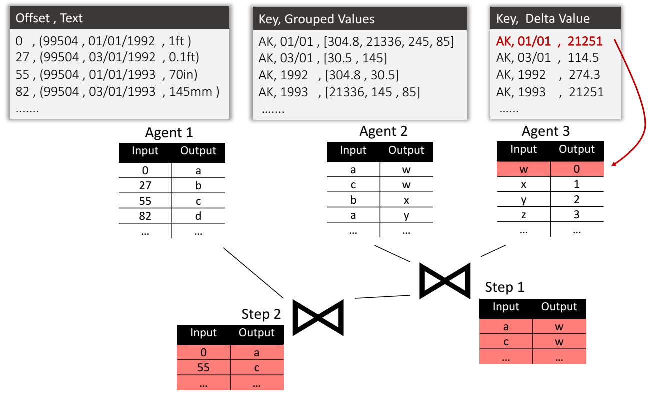

Offset , Text
0 , (99504 , 01/01/1992 , 1ft)
27 , (99504 , 03/01/1992 , 0.1ft)
55 , (99504 , 01/01/1993 , 70in)
82 , (99504 , 03/01/1993 , 145mm)
Key, Grouped Values
AK, 01/01 , [304.8, 21336, 245, 85]
AK, 03/01 , [30.5 , 145]
AK, 1992 , [304.8 , 30.5]
AK, 1993 , [21336, 145 , 85]
Key, Delta Value
AK, 01/01 , 21251
AK, 03/01 , 114.5
AK, 1992 , 274.3
AK, 1993 , 21251
Agent 1
Agent 2
Agent 3
Input
Output
Step 1
Step 2

Figure 4: A logical trace plan that recursively joins data lineage tables back to the input lines.

that maintains mappings between the input and output records. When a tracing query is issued, Titian recursively joins the agent tables as shown in Figure 4, which illustrates a trace from output record `AK , 01/01 , 21251` (id 0) to the records in the input file that derived it. Details on DAG instrumentation, distributed join of tracing tables, and API usage can be found in our previous paper [28]. For each faulty output, the corresponding fault-inducing input set from Titian is stored in a queue, C_L (line 5 in Algorithm 1).

Test Function Push Down. Spark applications comprise of hundreds to thousands of tasks running in parallel on different partitions. In the map-reduce programming paradigm, a *combiner* performs partial aggregation for operators such as `reduceByKey` on the map side before sending data to reducers to minimize network communication. Since Phase I uses a user-defined test function to check if each final record is faulty, our insight is that, during backward tracing, we should isolate the exact partitions with fault-inducing intermediate inputs to further reduce the backward tracing search scope.

Because faulty intermediate data records could have been already grouped together with non-faulty records from other partitions in an aggregation operation, we use the approach of pushing down a test-function to the earlier stage (i.e., combiner) to isolate fault-inducing partitions. In the intermediate stage where a test function could be moved to, BIGSIFT then determines which partitions are no longer relevant to faulty outputs and therefore obviates the need of tracing non-faulty partitions further.

Specifically, in Apache Spark, certain aggregation operators (*e.g.,* `reduceByKey`) require a user to provide an *associative* and *commutative* function as an argument. For a test function applied to these operators, BIGSIFT can push-down the user-defined test function to partitions in the previous stage to test intermediate results (line 2 in Algorithm 1) if the following three conditions are met: (1) the program ends with an aggregation operator (such as `reduceByKey`) that requires an associative function f_1; (2) $f_1 \circ f_2$ is associative, when f_2 is a test function; and (3) $f_1 \circ f_2$ is failure-monotone, which is analogous to the monotonicity assumption of DD, meaning that an inclusion of a failure-inducing intermediate record(s) in the partition produces a test failure, when combined with other intermediate data from other partitions. If these three requirements are not met,

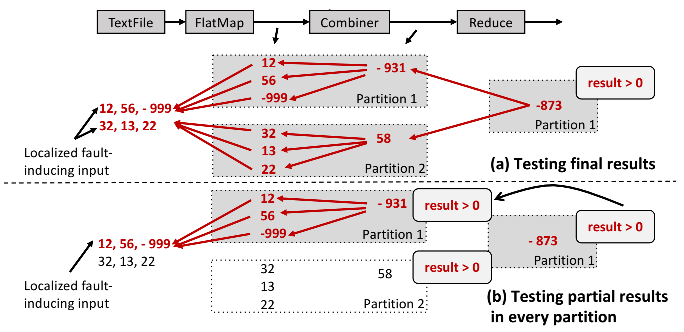


Figure 5: A decrease in the scope of potential fault-inducing input when the test function is pushed down. The workflow computes the sum of all the numbers in the input dataset.

```
sc.textFile(input)
  .flatMap{s => s.split(",").map(r => r.toInt)}
  .reduce( (a,b) => a+b )
  .collect()
```

Figure 6: A Spark program that computes the sum of all the numbers in the input dataset.

BIGSIFT rollbacks to using basic data provenance. Therefore, the applicability of test-driven provenance optimization does not affect debugging accuracy. Rather, when these conditions are met, BIGSIFT can speed up debugging time.

When the three conditions are met, $f_1 \circ f_2$ could be checked at each partition before the shuffle stage as a *combiner*, identifying faulty partitions early. For the individual partitions failing this combiner test function, BIGSIFT restricts backward tracing search only on those faulty partitions, significantly reducing the scope of potential fault-inducing input records. On the other hand, if the monotonicity property is not satisfied (which can be verified by testing the final output), or none of the partitions fail the test function, BIGSIFT rolls back to the default case of backward tracing using basic data provenance.

Figure 5 contrasts data provenance without vs. with this push down optimization on the program in Figure 6. This program computes the sum of all numbers in the input dataset. It first splits each line in the input into a list of numbers using `flatmap` and then uses an *add* function as a UDF for the `reduce` operator. Suppose that a user-provided test function checks whether the final output is greater than zero. Spark automatically inserts a combiner by pushing the test function `result=>(result>0)` to check the partial results from the combiner. Figure 5(b) shows that BIGSIFT checks the intermediate results of Partition 1 (on which the test fails) and restricts its backward tracing to only this partition, resulting in a significantly smaller subset of records. On the other hand, Figure 5(a) shows that, without this optimization, all partitions are considered for backward tracing.

4.2 Phase II: Prioritizing Backward Traces

Phase II applies to the case when we have multiple faulty output records to explain. It takes the set of backward traces as input and employs two prioritization heuristics *i.e., trace overlapping* and *smallest job first*, based on the insight that multiple failure symptoms could be caused by the same set of inputs. BigSift prioritizes backward traces to cover many faulty outputs within a time limit. When there is only one faulty output, Phase II is skipped.

Smallest Jobs First. Given multiple backtrace lineages from Phase I, BigSift prioritizes the trace with the smallest number of potential fault-inducing input records according to data provenance. This early discovery of fault-inducing input records may help users revise their code before other pending (larger) traces. Line 6 in Algorithm 1 sorts the backwards-trace queue in an ascending order.

Overlapping Backward Traces. Multiple faulty output records may be caused by the same input records due to operators such as `flatMap` or `join`, where a single data record can produce multiple intermediate records, leading to multiple faulty outputs. For example, in Figure 7, a fault-inducing input record `99504 , 01/01/1993 , 70in` generates more than one faulty output records, *i.e.,* `AK , 01/01 , 21251` (Figure 7(a)) and `AK , 1993 , 21251` (Figure 7(b)). While the cardinality of the individual backward trace from the faulty output is 4 and 3 respectively, the overlap of the two traces contains only two input records, leading to the two different faulty outputs (Figure 7(c)). The benefit of this prioritization is twofold. First, BigSift prioritizes the common input records leading to multiple outputs before applying DD to records that are pertinent to fewer faulty outputs. Second, the intersection of two sets might help us to tighten the scope of DD application, avoiding redundant work for the same failure-inducing records.

To check the eligibility for this optimization, BigSift explores the DAG of the Spark program to find at least one 1-to-many or many-to-many operator such as `flatMap` and `join`. The overlap is performed right after Phase II's "smallest job first", as shown by line 9 in Algorithm 1. BigSift overlaps the two smallest backward traces (let's say t_1 and t_2) from the sorted queue, C_L, to find the intersection, $t_1 \cap t_2$ (line 45). If the test function evaluated over the execution of $t_1 \cap t_2$ finds any fault, then DD is applied to $t_1 \cap t_2$ and the remaining (potential) failure-inducing inputs $t_1 - t_2$ and $t_2 - t_1$ (lines 47-48). Otherwise, DD is executed over both initial traces t_1 and t_2. If any fault-inducing inputs are found in the overlap, there could be potential time saving from not processing the overlap/intersection trace twice. Conversely, this prioritization could waste time for computing the overlap when the two backward traces do not overlap, or when the overlap trace does not cause any faulty output.

4.3 Phase III: Optimized Delta Debugging

Based on the order prioritized by Phase II, BigSift applies DD to each backward trace (lines 16-29 of Algorithm 1). BigSift provides a universal splitting function, which allows DD to deterministically split an input configuration into n sub-configurations (line 17). Each sub-configuration is then sequentially submitted for execution (line 18) until either a faulty sub-configuration is found (line 19), or all the sub-configurations pass the test (line 21). In the former case, DD is recursively called over the faulty sub-configuration. In the latter instead, each sub-configuration is used to compute a complement (lines 21-22) which are then executed and tested (line 23). If all the complements pass the test, DD either generates twice as many sub-configurations as before or n (size of original configuration)

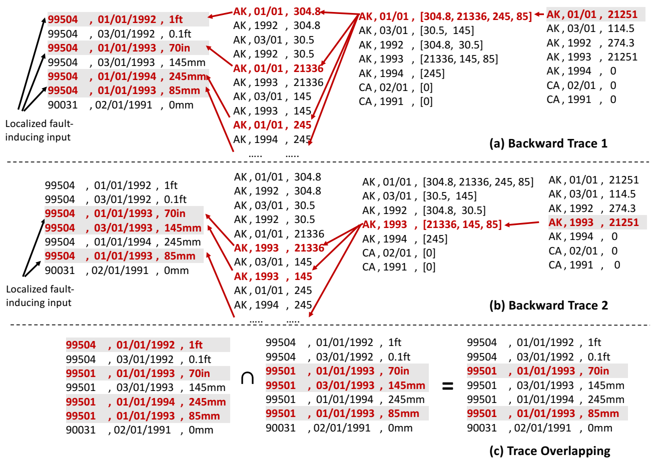


Figure 7: A decrease in fault-inducing input by overlapping backward traces of two faulty outputs emerging from the same fault-inducing input.

sub-configurations, which ever is smaller (line 27). It then starts testing these sub-configurations as explained earlier. Otherwise, if any one of the complement fails the test, DD starts exploring that sub-configuration (line 25).

Re-running a program on a large dataset can be extremely expensive. Next we describe two optimizations.

Bitmap Based Memoization of Test Results. In our running example from Figure 2(b), **Run 4** and **Run 7** test the same input configuration twice while applying delta debugging. DD is not capable of detecting redundant trials of the same input configuration and therefore tests the same input configuration multiple times. To avoid waste of computational resources, BigSift uses a *test results memoization* optimization. A naive memoization strategy would require scanning of the content of an input configuration to check whether it was tested already; such configuration content-based memoization would be time consuming and not scalable. BigSift instead leverages *bitmaps* to compactly encode the offsets in the original dataset, to refer to a sub-configuration.

The universal splitting function for DD is thus instrumented to generate sub-configurations along with their related bitmap descriptions. BigSift maintains the list of already executed bitmaps, each of which points to the test result of running a program on the input sub-configuration. Before processing an input sub-configuration, BigSift uses its bitmap description to perform a look-up in the list of bitmaps. If the result is positive, the test result for the target sub-configuration is directly reused by the look-up. Otherwise, BigSift tests the sub-configuration and enrolls its bitmap and the corresponding test result in the list (line 40 in Algorithm 1). This technique avoids redundant testing of the same input sub-configuration and reduces the total debugging time. BigSift uses the compressed Roaring Bitmaps representation to describe large scale datasets [34].

Adaptive Local Job Scheduling. When we investigate the debugging time spent for each run in DD and the number of input records, we discover that for a DD job small enough to be run on a single machine (e.g., less than 5000 records), running it on a cluster is

#	Subject Programs	Source	Input Size	# of Ops	Program Description	Input Data Description	Fault Location
S1	Movie Histogram	PUMA	30 GB	4	Counts the number of movies in each rating category using `map`, `reduceByKey`, and `filter`	Movies with corresponding ratings from raters	Code
S2	Inverted Index	PUMA	40 GB	5	Generates a word-to-document indexing of a text data using `flatmap`, `map`, and `reduceByKey`	Text data with corresponding file id	Code
S3	Rating Histogram	PUMA	30 GB	4	Generates the frequency of each rating score from raters using `flatmap`, `map`, and `reduceByKey`	Movies with corresponding ratings from raters	Code
S4	Sequence Count	PUMA	80 GB	5	Counts the occurrence of every 3-word sequence using `flatmap`, `map`, and `reduceByKey`	Text data from Wikipedia dump	Code
W1	Rating Frequency	Custom	30 GB	4	Counts the number of ratings from each rater using `flatmap`, `map`, and `reduceByKey`	Movies with corresponding ratings from raters	Code
W2	College Student	Custom	4 GB	4	Finds the average age of all the students per college year using `map` and `groupbykey`	Student data with name, year, and date of birth	Data
W3	Weather Analysis	Custom	20 GB	4	Finds, in each state, the delta between the minimum and maximum snowfall reading for each day of any year and for any particular year using `flatmap`, `map`, and `groupbykey`	Daily snowfall measurements for every zipcode in feet and millimeters	Data
W4	Transit Analysis	Custom	20 GB	4	Finds the total layover time of all passengers spending less than 45 minutes per airport and per hour using `map`, `filter`, and `reduceByKey`	Passenger's arrival and departure time along with the airport code and date	Code

Table 1: Subject programs with input datasets

unnecessary. BIGSIFT schedules a DD run on either the cluster or on a local machine, as shown in lines 36-39 in Algorithm 1.

5 EVALUATION

We perform a wide range of systematic experiments to evaluate BIGSIFT's runtime performance and precision of pinpointing fault-inducing input records compared against delta debugging and data provenance alone. To further differentiate the performance benefits from each optimization and prioritization, we design several versions of BIGSIFT as seen in Table 2: BIGSIFT-T simply combines delta debugging (DD) and test driven provenance (TP), BIGSIFT-O and BIGSIFT-S enable trace overlapping and smallest job first respectively in addition to leveraging both DD and TP. BIGSIFT-M applies bitmap based memoization of test results. Finally, BIGSIFT enables all optimization and prioritization heuristics. Our investigation addresses the following evaluation questions:

- How much improvement in the precision of fault-inducing input records does BIGSIFT provide in comparison to data provenance?
- How much improvement in the debugging time does BIGSIFT provide in comparison to delta debugging?
- When a time limit is set for fault localization, what are the benefits of *trace overlapping* and *smallest jobs first* prioritization heuristics respectively?

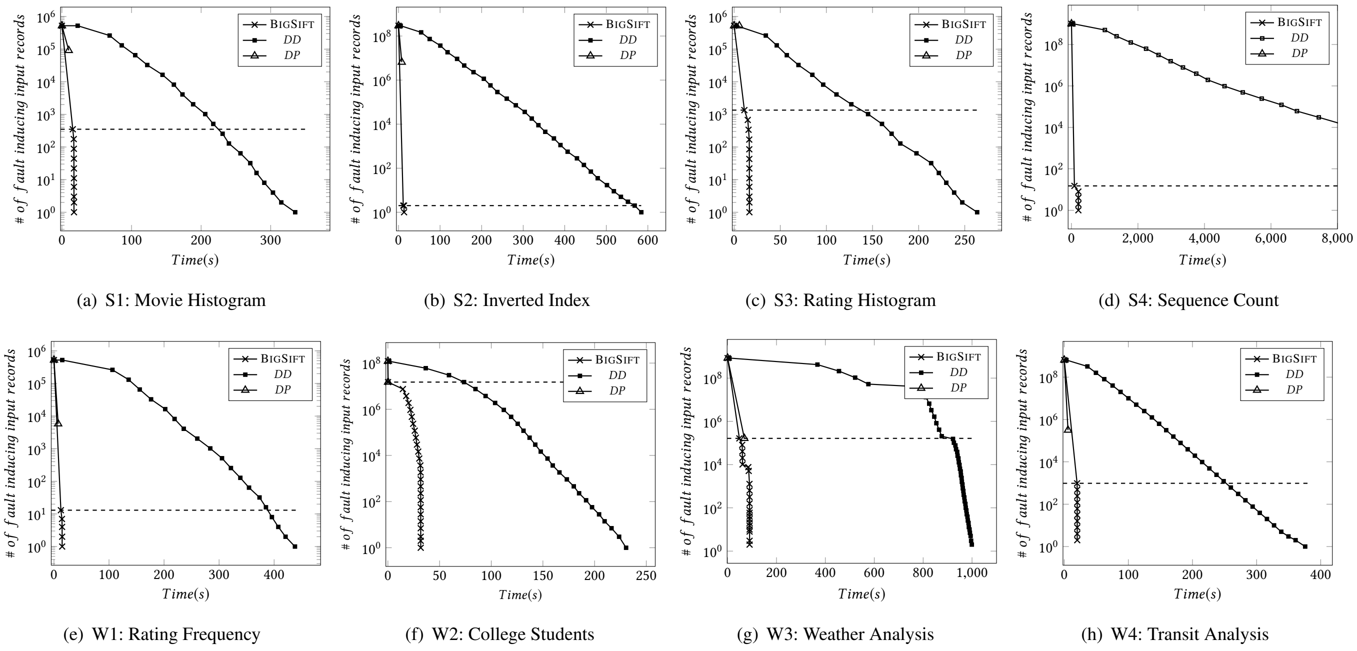
Evaluation Environment. We use a cluster consisting of sixteen i7-4770 machines, each running at 3.40GHz and equipped with 4 cores (2 hyper-threads per core), 32GB of RAM, and 1TB of disk capacity. The operating system is a 64bit Ubuntu 12.04. The datasets are all stored on HDFS version 1.0.4 with a replication factor of 3. The level of parallelism was set at two tasks per core. This configuration allows us to run up to 120 tasks simultaneously. BIGSIFT currently supports Apache Spark version 1.2.1 and leverages Titian to support data provenance in Spark. The runtime overhead of lineage capture from Titian is reported to be below 30% [28].

Name	DD	TP	Trace Overlap	SJF	MEM
BIGSIFT-T	✓	✓	✗	✗	✗
BIGSIFT-O	✓	✓	✓	✗	✗
BIGSIFT-S	✓	✓	✗	✓	✗
BIGSIFT-M	✓	✓	✗	✗	✓
BIGSIFT	✓	✓	✓	✓	✓

Table 2: BIGSIFT with various optimizations. TP, SJF, and MEM stand for test driven provenance, smallest job first, and test results memoization, respectively.

Subject Programs. We evaluate BIGSIFT using a comprehensive set of subject programs and custom real-world workflows. We use eight subject programs in total, four of which are adapted from MapReduce PUMA benchmark [4]. PUMA benchmark provides an extensive set of big data processing applications along with a large-scale dataset for Hadoop MapReduce frameworks. We also developed four custom Spark programs (W1) Rating Frequency, (W2) College Student Analysis, (W3) Weather Analysis, and (W4) Air Transit Analysis. Table 1 shows all the subject programs along with their description. All subject programs except W2, W3, and W4 use the dataset provided by PUMA Benchmark. In W2, W3, and W4 we generate our own datasets using data generation scripts whereas W1 uses the PUMA dataset.

Test Functions. Each of the subject programs is also accompanied with a test function that checks for the correctness of each output record. This is analogous to writing an assertion or a unit test case in software engineering. Knowing the validity of each output record does not necessarily mean that a user can identify a minimum subset of failure-inducing input records. For most programs, the test function checks if individual output records are within valid ranges. For example, the test functions for S3 and S4 check that the count is positive for each rating and for 3-word sequences respectively. As another example, the test function for W2 checks that the average age of students of each college year is between 16 and 26.



(a) S1: Movie Histogram (b) S2: Inverted Index (c) S3: Rating Histogram (d) S4: Sequence Count

(e) W1: Rating Frequency (f) W2: College Students (g) W3: Weather Analysis (h) W4: Transit Analysis

Figure 8: Performance comparison

Seeding Faults. The subject programs and their corresponding datasets used to evaluate BigSift do not contain any faults. Therefore, we either seed faulty data records in the input dataset or inject programming errors in the subject program's code. These two types of faults in our experiments underline the important distinction between data cleaning and debugging. Outliers or ill-formatted data records may be localized by intelligent data cleaning techniques; however, such data cleaning techniques cannot handle situations where the notion of faulty data keeps changing, depending on an application coding error. Given a test function, BigSift not only finds inconsistently formatted records in the input data but also isolates cleanly formatted records interacting with faulty code, resulting in faulty outputs. The last column shows whether a fault is injected in data vs. code.

To inject faulty data, we select a random input record and modify it differently for each input dataset. For example, in the case of weather telemetry data, we randomly pick a single input record and replace the value of the snow measurement with the value in the unit of inches. This fault affects the final output of W3 and fails a check that the delta snowfall reading should not exceed 6000 millimeters. Similarly, in the college student data analysis W2, the date of birth for a randomly selected student is mutated to the date "0/0/0", which leads to a test failure.

We introduce code faults by modifying program logic—*i.e.,* code faults are introduced in the user-defined function of a data transformation operator such that the program behaves differently for certain intermediate data records. For example, in S4, the map transformation is modified, so that whenever two 3-sequence words "He has also" and "Romeo and Juliet" appear together in a line, the count of "He has also" is replaced with -99999. Similarly, in the subject program W4, an injected code fault affects a small set of intermediate records leading to a wrong value for the delta between the

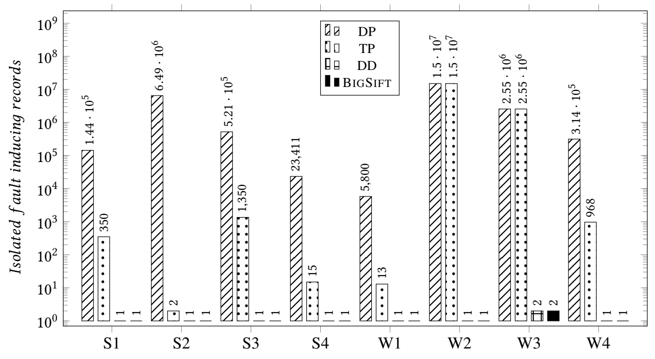
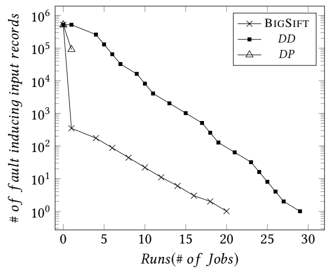


Figure 9: Fault localizability comparison

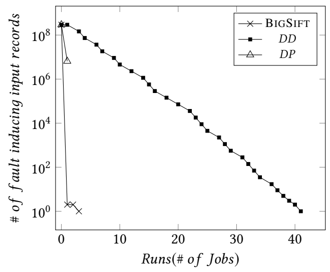
arrival and departure time of a passenger. For this case, the input data do not contain any data format anomaly or outliers. Six out of our eight subject programs contain code faults that cannot be debugged by data cleaning techniques because the notion of unclean data is dependent on coding faults.

5.1 Fault Localizability

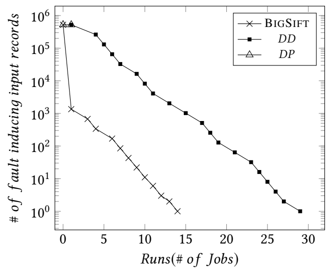
To evaluate the ability to precisely localize fault-inducing input records, we measure the final size of the fault-inducing inputs from BigSift. We also compare test-function driven data provenance (TP) with using data provenance (DP). The results are presented in Figure 9. The x-axis represents the subject programs, while the y-axis measures the number of fault-inducing input records from BigSift, DP, DD, and TP for each program. In almost all cases, data provenance over-approximates the fault-inducing input records, stopping at the order of 10^3 to 10^7 records, which is infeasible for programmers to manually sift through. For example, in program W2, DP is not able to localize fault-inducing input records beyond



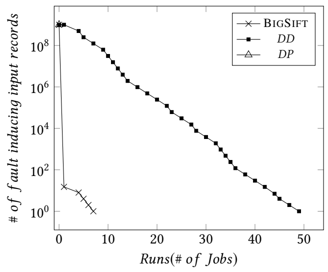
(a) S1: Movie Histogram



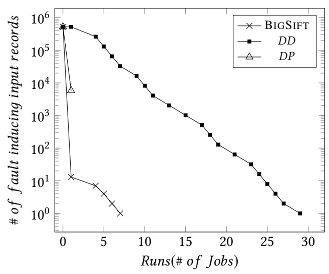
(b) S2: Inverted Index



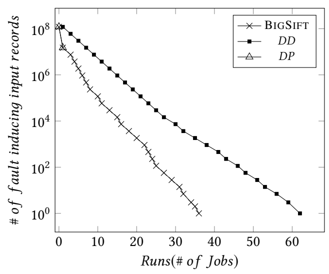
(c) S3: Rating Histogram



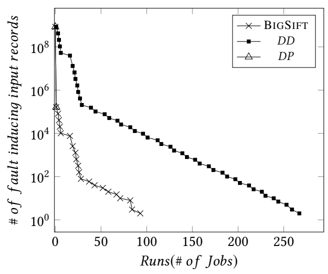
(d) S4: Sequence Count



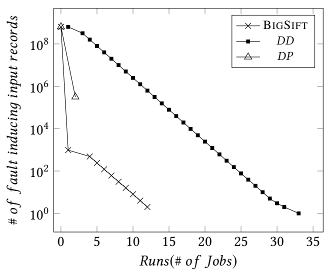
(e) W1: Rating Frequency



(f) W2: College Student



(g) W3: Weather Analysis



(h) W4: Transit Analysis

Figure 10: Reduction in the number of runs

15 million records. The poor localizability of DP is due to the use of `groupByKey` where the number of unique keys are only four possible keys, which results in over-approximating the scope of fault-inducing input records. On the other hand, we leverage test function push down in TP, when applicable, to reduce the size of fault-inducing input to a few thousand records (*e.g.,* in S1 and S3) by identifying faulty partitions (see dotted bars in Figure 9). BigSift leverages DD to continue fault isolation after TP, achieving even higher accuracy.

5.2 Debugging Time

To evaluate the performance improvement of BigSift, we compare the total debugging time of BigSift against the baseline delta debugging (DD) and data provenance (DP). At every single iteration of DD, we log three metrics—the number of program runs (*i.e.,* iterations), the number of the fault-inducing input records, and the corresponding time span. These metrics help us analyze the runtime behavior at a fine-grained level. We then apply BigSift on the same input data to localize the precise failure-inducing input records.

Figure 8 shows the performance improvement in BigSift compared to original DD. The x-axis represents the total debugging time in seconds and the y-axis represents the number of localized fault-inducing input records. For example, in Figure 8(d), BigSift takes 208 seconds to find the seeded fault, whereas DD takes 13772 seconds. DP stops after finding 23411 fault-inducing records in 398 seconds but cannot localize further from there. Comparison with DD shows that BigSift enhances the debugging time by 66X. Further analysis shows that by applying test function driven data provenance (TP), BigSift reduces the initial scope of fault-inducing records from more than 1 billion to just 15 records (dotted horizontal line) in 208 seconds, whereas DD takes 12395 seconds to achieve the same reduction. This significant decrease can be also seen in the other plots of Figure 8, as a steep drop till the dotted horizontal line, compared to the slow and steady elimination from DD marked in black. Figure 10 represents the number of runs required to perform fault localization. Figure 10(d) shows the result on S4. BigSift takes just 7 runs to reach the minimum fault-inducing records, while DD takes 49 runs to achieve the same.

Running Time (s)		Debugging Time (s)		
Program	Original Job	DD	BigSift	Improvement
S1	56.2	232.8	17.3	13.5X
S2	107.7	584.2	13.4	43.6X
S3	40.3	263.4	16.6	15.9X
S4	356.0	13772.1	208.8	66.0X
W1	77.5	437.9	14.9	29.5X
W2	53.1	235.2	31.8	7.4X
W3	238.5	999.1	89.9	11.1X
W4	45.5	375.8	20.2	18.6X

Table 3: Fault localization time improvement

Table 3 shows the overall reduction in debugging time in BigSift in comparison to DD. Overall, BigSift provides from up to a 66X speed up in the total debugging time, in comparison to DD. In the case where TP does not significantly reduce the size of the initial fault-inducing input, the speed up is 7.4X. Interestingly, *the time taken for automated debugging of a singly faulty output in* BigSift *on average is 62% less than the time taken for a single run on the entire data* (Columns Original Job vs. BigSift). With only up to 30% overhead incurred by Titian for lineage capture [28], BigSift dramatically reduces the scope and cost of iterative fault localization, by leveraging the lineage mappings.

The reason behind this feasibility of automatic debugging is that, in many subject programs, BigSift reduces the scope of fault-inducing input records by testing partially-aggregated results such that the later time spent on repetitive fault isolation in DD could be much smaller than the original time taken for the first run on the entire data. In fact, our result suggests that automated debugging can be brought to a reality more easily for data flow programs running in the DISC environments than other types of traditional C, C++, or Java applications, because debugging DISC workloads provide unique opportunities for systems-level optimizations.

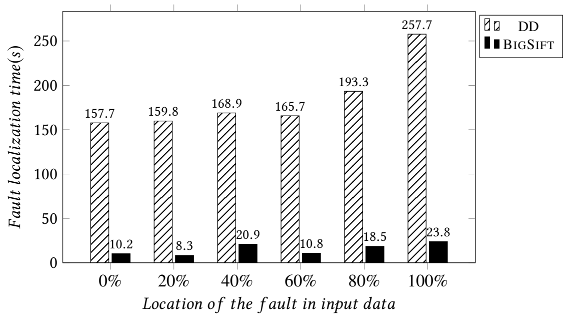


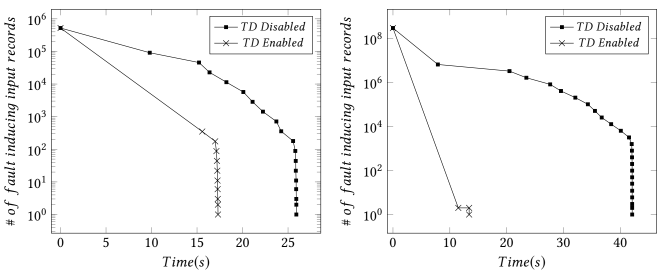
Figure 11: Fault localization time of BIGSIFT and DD for S1 w.r.t the location of seeded fault in input data.

Impact of Fault Location. While applying DD, the location of a faulty input record affects the total debugging time, because DD needs to test two sub-configurations sequentially at every iteration. If the first of the two always fails the test, DD will focus its search on the first one. Therefore, both DD and BIGSIFT may increase debugging time, if a fault-inducing record is located near the end of input data.

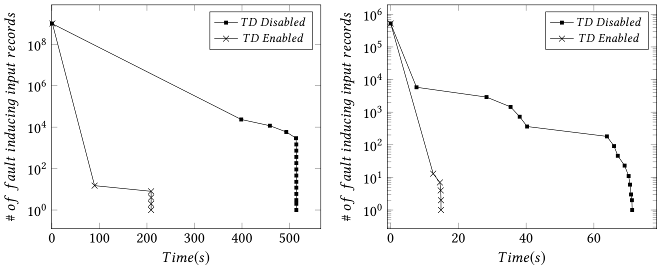
To evaluate the impact of fault-inducing input location on debugging time, we compare BIGSIFT with DD while varying the location of a fault-inducing input. Figure 11 summarizes the results where the x-axis represents the location of a fault (*e.g.,* 20% denotes that the fault is at one-fifth of the data) and the y-axis represents debugging time. When the location of fault-inducing input is changed from the start to the end (0% to 100%) with the increment of 20%, the debugging time of BIGSIFT increases from 10.2 seconds to 23.8 seconds for subject program S1. We also observe a similar trend in DD when the location of a fault is near the end of the input data.

Effects of Test Function Push Down. To evaluate the effects of test function push down (TD), we compare BIGSIFT with TD disabled vs. TD enabled. In the TD enabled version, BIGSIFT pushes down a user-defined test function to each individual partition to test partial results. Our evaluation targets subject programs whose last operator is `reduceByKey` (i.e., programs S1, S2, S3, S4, W1, and W4). For subject programs W2 and W3, the UDF of the last operator is not associative. For example, in W2, the last transformation computes the average of each group. Computing an average is a non-associative operation; therefore, TP becomes basic data provenance.

Figure 12 illustrates how BIGSIFT completes fault localization faster than BIGSIFT without TD. In subject program S1 (Figure 12(a)), BIGSIFT takes 17 seconds to localize fault-inducing input records, 33% less than BIGSIFT with TD disabled. By testing partial results and applying DP afterwards on faulty partitions, BIGSIFT reduces the scope of fault-inducing input to just 350 records, while disabling TD reduces the scope to 91308 records.



(a) S1: Movie Histogram (b) S2: Inverted Index



(c) S4: Sequence Count (d) W1: Rating Frequency

Figure 12: Effect of test function push down (TD).

5.3 Debugging Program Faults

As BIGSIFT is built on DD, by construction, it has the following characteristics:

- BIGSIFT does not enumerate all possible explanations. Instead, it finds a *single* minimum subset responsible for producing the same test failure. In other words, if there are two possible explanations of failure-inducing inputs, it finds one not both.
- BIGSIFT only guarantees to produce the same test failure when applying the given test function to the resulting set of fault-inducing input records. It may not produce the same faulty output value as the original failing run on the entire input.
- BIGSIFT is extremely beneficial for the case of finding *a needle in a haystack. i.e.,* both fault-inducing input and faulty output occur very rarely. Such debugging scenario is generally the most difficult case in software engineering, as developers cannot easily find a small, manageable size of data to reproduce the same failure symptom.

To manifest these strengths and limitations empirically, we design an experiment where we inject four different coding faults in subject program W4. These coding faults interact with a different amount of input records and produce different numbers of faulty output records in the final result. We compare performance and fault localizability with DD, DP, and the original running time.

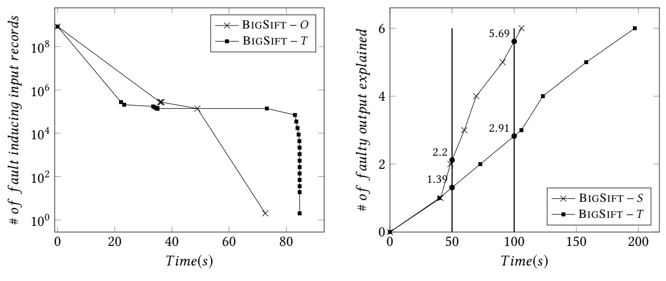
The program W4 calculates the total transit time of all passengers who spend less than 45 minutes at each airport grouped by every hour. The input dataset used by the program is completely clean *i.e.,* the dataset is free from any kind of formatting anomalies or outliers. The four different versions of W4 are listed in Table 4.

Program Versions	Original Job Time	Faulty Outputs	Data Affected	Debugging Time(s) BigSift	DP	DD	BigSift vs. Job Time	Fault Localization BigSift	DP	DD	Improvement
W4-1	48.8	1020	555464920	12442.0	19.5	>12 hr	255X	1020	283790715	1020	2.8x10^5X
W4-2	46.6	367	37642315	2658.5	9.3	>12 hr	57X	367	56789568	367	1.6x10^5X
W4-3	45.5	170	33320879	1144.1	8.8	>12 hr	25X	170	43318865	170	2.6x10^5X
W4-4	46.5	1	1	8.5	8.4	431	0.18X	1	84948	1	8.5x10^4X

Table 4: Performance and fault localization of BigSift on 4 versions of subject program W4 each with difference coding fault.

```
  val diff = input.map{ data =>
      val arr_min = getMinutes(data._2)
      val dep_min = getMinutes(data._3)
      var timediff = dep_min - arr_min
      //Branch removed to inject code fault
-     if(timediff < 0 ){
-       timediff = 24*60 + timediff
-     }
      timediff
   }
```

Figure 13: A branch is removed in subject program W4 to inject a code fault



(a) Trace Overlapping (b) Smallest Job First

Figure 14: Benefits from Trace Overlapping (a) and Smallest Job First (b)

We count the number of faulty output records using differential testing by comparing the final results of the faulty version against the original program. Depending on code faults, the number of faulty outputs ranges from 1 to 1020 faulty outputs (Faulty Outputs). We conservatively estimate the number of faulty input records by profiling individual input records exercised by the faulty code region. This number varies from a single record to several million records (Data Affected). Figure 13 shows an example code fault from program W4-3 that removes a code fragment, re-adjusting the transit period over midnight. When there are multiple faulty output records, we run BigSift and DD for each faulty output record in iteration. Table 4 summarizes the experiment results.

Consider the program version W4-1 that touches 555 millions input records and then generates 1020 faulty outputs. The entire process for debugging all 1020 faulty outputs takes 12442 seconds and the total time is 255X of the original job time. While the code fault touches 555 million input records, BigSift finds only 1020 faulty inputs, each of which corresponds to reproducing the test failure of a single faulty output. It is because the goal of Delta Debugging is to find a minimum set of fault-inducing records that can reproduce each test failure, not to enumerate all possible explanations for each failure.

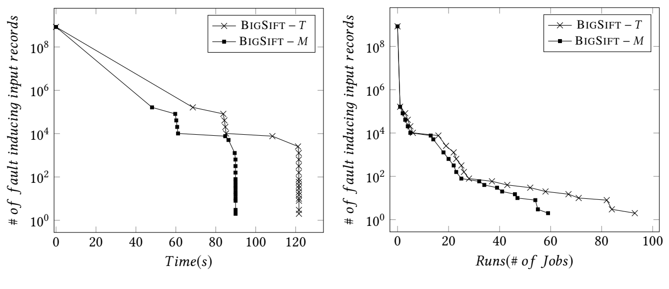
Nevertheless, BigSift still performs better than DD which will take an estimated 4 days ($\geq$ 100 hours) to find the equal number of fault-inducing inputs. In our experiments, we use a cut-off time of 12 hours for DD. DP finds more fault-inducing inputs than BigSift due to over-approximation, but the resulting set will also include non-faulty input data. For version W4-4, BigSift finds one and only fault-inducing input record precisely in 8.5 seconds, which 82% less than the original job time. This is the kind of *a needle in a haystack* situation where existing techniques take a very long time to debug. DP takes 8 seconds but fails to localize the fault-inducing input after reaching 84K records. The result shows that BigSift performs well in terms of localizing fault-inducing input and reducing debugging time, when both the affected inputs and faulty outputs are highly infrequent, which is often the most challenging case of debugging.

In practice, a developer is unlikely to use BigSift to provide all failure-inducing input records for all individual faulty outputs simultaneouly. Normally, a developer starts a debugging task with investigating the root cause of one faulty output and then fixes the source of the error before moving onto investigating the next faulty output. Therefore, in practice, when the program version W4-1 produces 1020 faulty outputs, we do not expect that a developer will run BigSift for 1020 iterations to find failure-inducing input records for individual faulty outputs all at once. In fact, several faulty output records (or several test failures) are often caused by a single code fault or similar data faults. Therefore, a fix for a single fault may remove more than one faulty output. The results summarized above illustrate a very conservative and exceptional debugging scenario, where a developer wants to find the source of all individual faulty outputs at once without fixing the discovered errors along the way.

5.4 Optimization and Prioritization Effect

To assess the benefits from each optimization and prioritization heuristic, we design four different versions of BigSift as illustrated in Table 2. Each variation was evaluated on all subject programs, unless not applicable.

Trace Overlapping. We evaluate *trace overlapping* in BigSift-O to assess its prioritization benefit, when there are multiple faulty output records. The benefits of overlapping the traces is debugging time reduction by prioritizing the common failure-inducing inputs that may be responsible for multiple faulty outputs. To assess whether this prioritization achieves any time saving, we compare BigSift-O vs. BigSift-T on subject programs W3, where we have 2 faulty output records. This program includes a `flatMap` operator which propagates a single faulty record into multiple faulty records. We observe (1) the number of fault-inducing input records identified within the same time limit and (2) the overall improvement in debugging time. In Figure 14(a), BigSift-O produces the exact same



(a) Fault Localization Time (b) Number of Runs

Figure 15: Benefits of Bitmap Based Memoization w.r.t fault localization time (a) and number of DD runs (b)

set of fault-inducing input records in 86% of the time, compared to BIGSIFT-T, by saving the time to identify the 1158 overlapping failure-inducing records twice. Figure 14(a) shows that BIGSIFT-O incurs an initial cost of computing the intersection. However, the remaining of the two overlapped traces do not contain the fault, which saves the fault localization time by not applying DD on them. The benefit of this prioritization is notable especially TD is not applicable.

Smallest Jobs First. BIGSIFT-S prioritizes backward traces in a *smallest job first* manner in an ascending order of the cardinality of backward traces from data provenance. This prioritization improves the coverage of faulty outputs when there are multiple faulty outputs to explain within the same time limit. We compare the coverage of the faulty outputs with BIGSIFT-S and BIGSIFT-T on program W3 with 6 faulty output records, where BIGSIFT-T selects traces at random.

Figure 14(b) illustrates the comparison. The y-axis represents the number of faulty output records explained, and the x-axis represents the time spent to perform these tasks. The reference lines at 50 and 100 seconds represent different possible time limits. By prioritizing DD on the cardinality of the scope of potential failure-inducing input records, BIGSIFT-S explains 5 faulty output records in W3, whereas the baseline BIGSIFT-T explains only 2 faulty output records with 100 seconds as the time limit.

Bitmap Based Memoization of Test Results. When applying DD in Phase III, in order not to test the same input sub-configuration multiple times, BIGSIFT-M uses the *test results memoization* optimization by maintaining a list of configuration descriptions (bitmaps) and the corresponding test outcomes.

To evaluate the advantage of this optimization, we compare BIGSIFT-M with BIGSIFT-T on program W3. Figure 15(b) shows the comparison in terms of the number of DD runs where the x-axis represents the number of jobs executed and the y-axis represents the size of the fault-inducing input set. BIGSIFT-M eliminates 34 duplicate tests in Phase III by caching test results. BIGSIFT-M needs 59 runs to find the minimum fault-inducing input, whereas BIGSIFT-T needs 93 runs to get the same result. The savings with respect to DD runs is also reflected as reduction in the debugging time of BIGSIFT-M. Figure 15(a) shows that BIGSIFT-M takes 89 seconds as opposed to 121 seconds for BIGSIFT-T to localize the minimum fault-inducing input. On program W3, test memoization reduces the debugging time by 26%.

6 RELATED WORK

Data dependence analysis for fault detection. Detecting bugs in the input by analyzing data dependence has been well explored both in *software engineering* and *databases*. In the database field, data provenance (also known as data lineage) is a tool used to explain how query results are related to input data [17]. Data provenance has been successfully applied both in scientific workflows and databases [6, 7, 17, 24]. RAMP [26] and Newt [36] add data provenance support to DISC systems; both are capable of performing backward tracing of faults to failure-inducing inputs. However, as our experiments show, data provenance alone is often not able to compute the minimum input failure-inducing set.

Ikeda et al. present provenance properties such as minimality and precision for individual transformation operators to support data provenance [25, 27]. However, their definition of minimality (minimum provenance) is based on reproducing the same output record rather than producing a faulty output. Therefore, their technique does not guarantee a minimum set of fault-inducing inputs. In the domain of network diagnosis, DiffProv analyses the differences between a provenance tree leading to a bad event and the other leading to a good event [10]. However, this approach requires a user to come up with a pair of a correct input and a failure-inducing input, very similar to each other. Finding such pair is extremely hard in DISC applications, because a user must synthesize two different input files, producing similar but not identical intermediate results in each stage. Chothia et al. [12] is a provenance system implemented over a differential dataflow system, like Naiad [40]. Their approach is more focused on how to provide semantically correct explanations of outputs through replay by leveraging the properties of a differential dataflow system.

In software engineering, dynamic taint analysis utilizes information flow analysis detect security bugs (*e.g.,* [37, 41]) and is also used to perform software testing and debugging (*e.g.,* [14, 33]). For example, Penumbra leverages dynamic taint analysis to automatically identify failure-relevant inputs [15]. It requires fine-grained tagging of program variables to track their flow in a program execution which can tremendously slow down the processing of DISC applications. Furthermore, it also suffers from the same limitation of over-approximating failure-relevant inputs and thus requires manual investigation. Program slicing is another technique that isolates statements or variables involved in generating a certain faulty output [3, 23, 47]. These techniques use either static and dynamic approaches to localize relevant code regions. Chan et al. identify failure-inducing input data by leveraging dynamic slicing and origin tracking [9]. Due to a large amount of data in DISC, tracing input records over program statements would be costly. BIGDEBUG is an interactive debugger for Spark [20–22]. Just like any interactive debuggers such as `gdb`, it is left to the developer to control the debugger to identify the root cause of errors. On the other hand, BIGSIFT performs automated debugging, when a test function is provided. BIGDEBUG's crash culprit feature uses basic data provenance to find the subset of input data causing a crash. BIGSIFT overcomes this very limitation of over-approximating crash-inducing input records using data provenance only by leveraging delta debugging, test-function push down, and other optimizations in tandem. A technique similar to test-function push down was previously used in [30] in the context of improving program re-execution performance after a bug fix.

Automated debugging through systematic experiments. Delta debugging is a well known technique for finding the minimal failure-inducing input that causes the program to fail [51], and has been used for a variety of applications to isolate the cause-effect chain or fault-inducing thread schedules [11, 16, 50]. As stated earlier, DD requires multiple executions of the program, which alone, is not tractable for DISC system workloads. HDD tries to minimize the number of executions involved in DD under the assumption that the input is in a well defined hierarchical structure [39]. In the DD split phase, HDD eliminates invalid configurations resulting in fewer runs. Thus HDD can reduce debugging time for hierarchically structured input data such as a HTML or XML document. In our context, this assumption rarely holds because the input dataset is often not hierarchically structured.

Intervention-based explanation Systems. Several systems have recently addressed the limitations of traditional data provenance to explain anomalous results by computing subsets of the lineage having an "influence" on the outlier result. Systems of this category delete candidate solutions, *i.e.,* groups of tuples, from the input and evaluate whether the outlier has changed. This process is called intervention and it is iteratively repeated in order to find the most influential groups of tuples, usually referred to as explanations [38, 42, 48]. Meliou et al. pioneer this research area by studying causality in the database area. They identify tuples, seen as potential causes, that are responsible of answers and non-answers to queries [38]. To pursue this task, they introduce the degree of responsibility to measure how responsible these tuples are. Scorpion finds outliers in the dataset that have the most influence on the final outcome [48]. It restricts itself to queries with aggregates over singles tables (*i.e.,* no joins are involved). Carbin et al. solve the similar problem of finding the influential (critical) regions in the input dataset that have higher impact on the output using fuzzed input, execution traces, and classification [8]. Roy et al. mix intervention with causality to overcome the limit of Scorpion in generating explanations over a single table only [42]. Finally, Data X-ray [46] extracts a set of features representing input data properties and summarizes the errors in a structured dataset. It considers the properties of data only, and does not reason about how a given program takes the input records and outputs faulty output records.

The goal of these explanation systems is similar to ours. While they focus on finding tuples that maximize the influence over a set of records of interest, our goal is to generate the minimal failure-inducing records. Different from BIGSIFT, these systems target specific set of queries and structured data, and therefore are not applicable to generic programs containing, for example, arbitrary UDFs. Furthermore, these approaches are commonly coupled with a DBMS, which hence limit their scalability.

Data cleaning. Input fault localization over structured data is related to the field of data cleaning. In traditional data cleaning, a set of specific user-defined rules is used to determine a set of constrains determining data errors when violated [13, 19, 29, 45]. In contrast to BIGSIFT, these approaches are independent from any subject program and its test function. The drawback is that defining all possible input data errors upfront is a daunting task even for a domain expert. Additionally, rule systems are not scalable for debugging purposes because they mostly run on centralized servers (a notable exception being [32]).

7 CONCLUSION AND FUTURE WORK

We are in the early days of debugging big data analytics. This paper presents the first automated debugging toolkit that combines insights from both data provenance in the database systems community and iterative systematic fault isolation in the software engineering community. Our experiments show automated debugging can be done in a *scalable* and *precise* manner by leveraging the semantics of data flow operators, the properties of data partitioning, and test data provenance, to reduce the scope of failure-inducing records up front, before initiating an optimized delta debugging.

Our experimental results highlight and motivate further opportunities for big data debugging. For example, finding failure-inducing inputs is just the beginning, but it is important to generalize the characteristics from the resulting set of failure-inducing inputs to automatically construct a data cleaning program. As another example, to identify failure-inducing code regions to be repaired, we must contrast the coverage profile of failure-inducing inputs against the coverage profile of success-inducing inputs using techniques such as spectra-based fault localization [31]. Additionally, we seek new cost-based optimizations for DISC systems that gather statistics at runtime to optimize repetitive DD workloads.

Acknowledgments. We thank the anonymous reviewers and Asim Kadav for their comments. Participants in this project are in part supported through NSF-1460325, NSF-12394018, IIS-1302698, CNS-1351047, and NIBIB-U54EB020404. We would also like to thank our industry partners at Google and Intel for their gifts.

REFERENCES

[1] Hadoop. http://hadoop.apache.org/.

[2] Spark. https://spark.apache.org/.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.

[4] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar. Puma: Purdue mapreduce benchmarks suite. Technical report, School of Electrical and Computer Engineering, Purdue University, 2012 . TRECE-12-11.

[5] E. Ainy, P. Bourhis, S. B. Davidson, D. Deutch, and T. Milo. Approximated summarization of data provenance. In *CIKM*, pages 483–492, 2015.

[6] M. K. Anand, S. Bowers, and B. Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 287–298, New York, NY, USA, 2010. ACM.

[7] O. Biton, S. Cohen-Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 1072–1081, Washington, DC, USA, 2008. IEEE Computer Society.

[8] M. Carbin and M. C. Rinard. Automatically identifying critical input regions and code in applications. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 37–48, New York, NY, USA, 2010. ACM.

[9] T. W. Chan and A. Lakhotia. Debugging program failure exhibited by voluminous data. *Journal of Software Maintenance*, 1998.

[10] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 115–128, New York, NY, USA, 2016. ACM.

[11] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 210–220, New York, NY, USA, 2002. ACM.

[12] Z. Chothia, J. Liagouris, F. McSherry, and T. Roscoe. Explaining outputs in modern data analytics. *Proc. VLDB Endow.*, 9(12):1137–1148, Aug. 2016.

[13] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *Proc. VLDB Endow.*, 6(13):1498–1509, Aug. 2013.

[14] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[15] J. Clause and A. Orso. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 249–260, New York, NY, USA, 2009. ACM.

[16] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM.

[17] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *The VLDB Journal*, 12(1):41–58, May 2003.

[18] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[19] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *The VLDB Journal*, 21(2):213–238, Apr. 2012.

[20] M. A. Gulzar, M. Interlandi, T. Condie, and M. Kim. Bigdebug: Interactive debugger for big data analytics in apache spark. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1033–1037, New York, NY, USA, 2016. ACM.

[21] M. A. Gulzar, M. Interlandi, T. Condie, and M. Kim. Debugging big data analytics in spark with bigdebug. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1627–1630, New York, NY, USA, 2017. ACM.

[22] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein, and M. Kim. Bigdebug: Debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 784–795, New York, NY, USA, 2016. ACM.

[23] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 263–272, New York, NY, USA, 2005. ACM.

[24] T. Heinis and G. Alonso. Efficient lineage tracking for scientific workflows. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1007–1018, New York, NY, USA, 2008. ACM.

[25] R. Ikeda, J. Cho, C. Fang, S. Salihoglu, S. Torikai, and J. Widom. Provenance-based debugging and drill-down in data-oriented workflows. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1249–1252, April 2012.

[26] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *In Proc. Conference on Innovative Data Systems Research (CIDR)*, 2011.

[27] R. Ikeda, A. D. Sarma, and J. Widom. Logical provenance in data-oriented workflows? In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 877–888, April 2013.

[28] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie. Titian: Data provenance support in spark. *Proc. VLDB Endow.*, 9(3):216–227, Nov. 2015.

[29] M. Interlandi and N. Tang. Proof positive and negative in data cleaning. In *2015 IEEE 31st International Conference on Data Engineering*, pages 18–29, April 2015.

[30] M. Interlandi, S. D. Tetali, M. A. Gulzar, J. Noor, T. Condie, M. Kim, and T. Millstein. Optimizing interactive development of data-intensive applications. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 510–522, New York, NY, USA, 2016. ACM.

[31] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

[32] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and S. Yin. Bigdansing: A system for big data cleansing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1215–1230, New York, NY, USA, 2015. ACM.

[33] T. R. Leek, G. Z. Baker, R. E. Brown, M. A. Zhivich, and R. Lippmann. Coverage maximization using dynamic taint tracing. Technical report, DTIC Document, 2007.

[34] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Softw. Pract. Exper.*, 46(11):1547–1569, Nov. 2016.

[35] F. Li and S. Nath. Scalable data summarization on big data. *Distributed and Parallel Databases*, 32(3):313–314, 2014.

[36] D. Logothetis, S. De, and K. Yocum. Scalable lineage capture for debugging disc analytics. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 17. ACM, 2013.

[37] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *15th International Symposium on Software Reliability Engineering*, pages 198–209, Nov 2004.

[38] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *PVLDB*, 4(1):34–45, 2010.

[39] G. Misherghi and Z. Su. Hdd: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 142–151, New York, NY, USA, 2006. ACM.

[40] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[41] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *In In Proceedings of the 12th Network and Distributed Systems Security Symposium*. Citeseer, 2005.

[42] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *SIGMOD*, pages 1579–1590, 2014.

[43] J. Shafer, S. Rixner, and A. L. Cox. The hadoop distributed filesystem: Balancing portability and performance. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 122–133. IEEE, 2010.

[44] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990.

[45] J. Wang and N. Tang. Towards dependable data repairing with fixing rules. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 457–468, New York, NY, USA, 2014. ACM.

[46] X. Wang, X. L. Dong, and A. Meliou. Data x-ray: A diagnostic tool for data errors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1231–1245, New York, NY, USA, 2015. ACM.

[47] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[48] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *Proc. VLDB Endow.*, 6(8):553–564, June 2013.

[49] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[50] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA, 2002. ACM.

[51] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.