Influence-Based Provenance for Dataflow Applications with Taint Propagation

Jason Teoh  
UCLA  
jteoh@cs.ucla.edu

Muhammad Ali Gulzar  
Virginia Tech  
gulzar@cs.vt.edu

Miryung Kim  
UCLA  
miryung@cs.ucla.edu

ABSTRACT
Debugging big data analytics often requires a root cause analysis to pinpoint the precise culprit records in an input dataset responsible for incorrect or anomalous output. Existing debugging or data provenance approaches do not track fine-grained control and data flows in user-defined application code; thus, the returned culprit data is often too large for manual inspection and expensive post-mortem analysis is required.

We design FLOWDEBUG to identify a highly precise set of input records based on two key insights. First, FLOWDEBUG precisely tracks control and data flow within user-defined functions to propagate taints at a fine-grained level by inserting custom data abstractions through automated source to source transformation. Second, it introduces a novel notion of influence-based provenance for many-to-one dependencies to prioritize which input records are more responsible than others by analyzing the semantics of a user-defined function used for aggregation. By design, our approach does not require any modification to the framework’s runtime and can be applied to existing applications easily. FLOWDEBUG significantly improves the precision of debugging results by up to 99.9 percentage points and avoids repetitive re-runs required for post-mortem analysis by a factor of 33 while incurring an instrumentation overhead of 0.4X - 6.1X on vanilla Spark.

CCS CONCEPTS
• Information systems → MapReduce-based systems; • Theory of computation → Data provenance; • Software and its engineering → Software testing and debugging.

KEYWORDS
Big data systems, data intensive scalable computing, data provenance, fault localization, taint analysis

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ACM Reference Format:

1 INTRODUCTION
As the capacity to store and process data has increased remarkably, large scale data processing has become an essential part of software development. Data-intensive scalable computing (DISC) systems, such as Google’s MapReduce [17], Hadoop [2], and Apache Spark [3], have shown great promises to address the scalability challenge.

The correctness of DISC applications depends on their ability to handle real-world data; however, data is inherently incomplete, continuously evolving, and hard to know a priori. Erroneous or invalid data could cause failures in a data processing pipeline or produce a wrong output. Developers then need to identify the exact records responsible for such errors and failures by distinguishing a critical set of input records from billions of other records.

To address this problem of identifying the root cause of a wrong result or an application failure, data provenance techniques [15, 25, 30] capture input-output record mappings at each transformation level (e.g., map, reduce, join) at runtime and enable backward tracing on a suspicious output. However, these techniques suffer from two fundamental limitations. First, because these techniques capture input-to-output mappings only at the level of dataflow operators and do not analyze the internal semantics of user-defined functions (UDFs) passed to these operators, the backward trace tends to be much larger and contains input records that may not logically connected to the output under investigation. For example, input.map(s:List[Int] => s.max) takes a set of collections as input and applies the UDF on each collection, propagating only the maximum value to the succeeding operation. An existing data provenance technique such as Titian [25] would consider all elements in the collection relevant to the returned max value. Second, during an aggregation operation such as reduce, even though the degree of

* This research was done while the second author was a graduate student at UCLA.
influence towards an aggregated output differs among individual inputs depending on the semantics of UDFs, existing provenance techniques treat UDFs as a black box and naively consider that the aggregated output depends on all inputs without any priority or selectivity. Suppose that an aggregation operation calculates the standard deviation of a set of numbers. When a user gets a suspiciously high standard deviation value, she would like to isolate the most influential record(s) farther away from the mean, instead of the entire input data. Therefore, they are incapable of identifying the most relevant subset of inputs that contribute to the suspicious output and thus return a significantly larger backward trace.

Alternatively, search-based debugging techniques [18, 43] can be used for post-mortem analysis as they repetitively run the program with different input subsets and check whether a test failure appears. Thus, these black-box techniques require multiple re-runs with different input subsets, which can take several hours, if not days.

In light of these limitations of existing approaches, we provide the first influence-based debugging tool for data intensive scalable computing applications, called FLOWDEBUG. Given a suspicious output, it identifies the precise record(s) that contributed the most towards generating the suspicious output for which a user wants to investigate its origin.

The key idea of FLOWDEBUG is twofold. First, FLOWDEBUG incorporates white-box tainting to account for the effect of control and data flows in UDFs, all the way to individual variable-level in tandem with traditional data provenance. This fine-grained taint analysis is implemented through automated transformation of a DISC application by injecting new data types to capture logical provenance mappings within UDFs. Second, to drastically improve both performance and utility of identified input records, FLOWDEBUG incorporates the notion of influence functions [27] at aggregation operators to selectively monitor the most influential input subset. FLOWDEBUG predefines influence functions for commonly used UDFs, and a user can also provide custom influence functions to encode their notion of selectivity and priority suitable for the specific UDF passed as an argument to the aggregation operator. Taint analysis by definition incurs extra overhead, while influence functions attempt to reduce the size of input-output mappings for aggregations. To our knowledge, no prior work combines both approaches to improve debugging precision.

While existing data provenance techniques modify the runtime of DISC frameworks, FLOWDEBUG does not require any modifications to the framework’s runtime and instead provides an API on top of existing data structures such as Apache Spark RDDs, making it easier to adopt. Other data provenance approaches that leverage the notion of influence [10, 40] or taint analysis [38] are limited in their generalizability, because they either rely on predefined, operator-specific data-partition strategies or require the costly practice of intercepting billions of system calls to process taint marks.

We evaluate FLOWDEBUG on three primary research evaluation questions related to precision, recall, and performance and compare it with state-of-the-art data provenance and search-based debugging techniques. Compared to Titian [25], FLOWDEBUG improves precision by up to 99.9 percentage points. Compared to BigSift [18], FLOWDEBUG is able to improve recall by up to 99.3 percentage points. Finally, FLOWDEBUG is able to perform debugging up to 51X faster than Titian and 1000X faster than BigSift while adding an instrumentation overhead of 0.4X - 6.1X compared to Apache Spark. FLOWDEBUG shows remarkable improvement in today’s automated debugging of large-scale data processing and reduces manual human effort in a root cause analysis.

2 MOTIVATING EXAMPLE

This section discusses two examples of Apache Spark applications, inspired by the motivating example presented elsewhere [18], to show the benefit of FLOWDEBUG. FLOWDEBUG targets commonly used big data analytics running on top of Apache Spark, but its key idea generalizes to any big data analytics running on data intensive scalable computing (DISC) frameworks.

Suppose we want to analyze a large dataset that contains weather telemetry data in the US over several years. Each data record is in a CSV format, where the first value is the zip code of a location where the snowfall measurement was taken, the second value marks the date of the measurement in the mm/dd/yyyy format, and the third value represents the measurement of the snowfall taken in either feet (ft) or millimeters (mm). 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
```

2.1 Running Example 1

Consider an Apache Spark program, shown in Figure 1 that performs statistical analysis on the snowfall measurements. For each state, the program computes the largest difference between two snowfall readings for each day in a calendar year and for each year. Lines 5-18 show how each input record is split into two records: 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. We use function `convertToMm` at line 10 of Figure 1a to normalize all snowfall measurements to millimeters. Similarly, we uses `zipToState` at line 7 to map zipcode to its corresponding state. To measure the biggest difference in snowfall readings (Figure 1), we group the key value pairs using `groupByKey` in line 24, yielding
val log = "s3n://xcr:wYgws/logs/weather.log"
val input :RDD[String] = new 
SparkContext(sc).textFile(log)
val split = input.flatMap{ s: String =>
val tokens = s.split("\","")
// finds the state for a zip code
var state = zipToState(tokens(0))
var date = tokens(1)
// gets snow value and converts it into millimeter
val snow = convertToMm(tokens(2))
// gets year
val year = date.substring(date.lastIndexOf("/"))
// gets month / date
val monthdate= date.substring(0,date.lastIndexOf("/"))
List[((String, String), Float)] =
((state , monthdate), snow), ((state , year), snow)

val snowfall = split.onAny
groupByKey().mapValues{ s: List[Float] =>
s.max - s.min
}
val deltaSnow = split.groupByKey().mapValues{ s: List[Float] =>
s.max - s.min
val deltaSnow.saveAsTextFile("hdfs://s3-92:9010/*")

Figure 1: Example 1 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 27. The red rectangle highlights code edits to enable FlowDebug's UDF-aware taint propagation. Although Scala does not require explicit types to be declared, some variable types are mentioned in orange color to highlight type difference.

Figure 2: A filter function that searches for input data records with more than 6000mm of snowfall reading.

After running the program in Figure 1a and inspecting the result, the programmer finds that a few output records have suspiciously high delta snowfall values (e.g., AK, 1993, 2125). To trace the origin of these high output values, suppose that the programmer performs a simple scan on the entire input to search for extreme snowfall values using the code shown in Figure 2. However, such scan is unsuccessful, as it does not find any obvious outlier.

An alternative approach would be to isolate a subset of input records contributing to each suspicious output by using search-based debugging [18] or data provenance [25], both of which have limitations related to inefficiency and imprecision, discussed below.

Imprecision of Data Provenance. Data provenance is a popular technique in databases. It captures the input-output mappings of a data processing pipeline to explain the output of a query. In DISC applications, these mappings are usually captured at each transformation level (e.g., map, reduce, join) [25] and then backward recursive join queries are run to trace the lineage of each output record. Most data provenance approaches [5, 6, 15, 21, 23, 25, 30] are coarse-grained in that they do not analyze the internal control flow and data flow semantics of user-defined functions (UDFs) passed to each dataflow operator and by treating them as a black box. Thus, they overestimate the scope of input records related to a suspicious output. For example, Titan would return all 6,063,000 input records that belong to the key group...
(AK, 1993). even though the UDF passed to mapValues in line 24 of Figure 1a uses only the maximum and minimum values within each key group to compute the final output. While Titian used the term fine-grained provenance to refer to its record-level provenance, we redefine the term to refer to more precise, fine-grained provenance that is both record-level and UDF-semantics aware. 

**Inefficiency of Search-based Debugging.** Delta Debugging (DD) [41] is a well known search-based debugging technique that eliminates irrelevant inputs by repetitively re-running the program with different subsets of inputs and by checking whether the same failure is produced. In other words, narrowing down the scope of responsible inputs requires repetitive re-execution of the program with different inputs. For example, BigSift [18] would incur 41 runs for Figure 1a, since its black-box debugging procedure does not recognize that the given UDF at line 26 selects uses only two values (min and max) for each key group.

**Debugging Example 1 with FLOWDEBUG.** To enable FLOWDEBUG, we replace SparkContext with FlowDebugContext that exposes a set of ProvenanceRDD, enabling both influence-based data provenance and taint propagation. Figure 3 shows this automatic type transformation and the red box in Figure 1b highlights those changes. Instead of textFile which returns an RDD of type String, we use textFileWithTaint to read the input data as a ProvenanceRDD of type TaintedString. The UDF in Figure 1a lines 5-18 now expects TaintedString as input and returns a list of tuple with tainted primitive types. Although a user does not need to explicitly mention the variable types due to compile-time type inference in Scala, we include them to better illustrate the changes incurred by FLOWDEBUG.

The use of FlowDebugContext also triggers automated code transformation to refactor the input/return types of any method used within a UDF such as convertToMm at line 27 of Figure 1b. At runtime, FLOWDEBUG uses tainted primitive types to attach a taint object to the primitive type for provenance tracking purposes. By doing so, FLOWDEBUG can track the provenance inside the UDF and improves precision. For example, the UDF at line 24 of Figure 1b performs selection with min and max on the input list. Since the data type of input list (s) is List[TaintedFloat], FLOWDEBUG propagates the provenance of only the minimum and maximum elements in the list. The final outcome contains the list of references of the following records that are responsible for a high delta snowfall.

<table>
<thead>
<tr>
<th>Row</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>textFile</td>
<td>Row: String</td>
</tr>
<tr>
<td>RDD</td>
<td>Data: String</td>
</tr>
<tr>
<td>textFileWithTaint</td>
<td>ProvenanceRDD</td>
</tr>
<tr>
<td>RDD</td>
<td>Data: TaintedString</td>
</tr>
<tr>
<td>key.(String,String)</td>
<td>Value: Float</td>
</tr>
<tr>
<td>ProvenanceRDD</td>
<td>Key: (TaintedString, TaintedString)</td>
</tr>
<tr>
<td>Value: TaintedFloat</td>
<td></td>
</tr>
<tr>
<td>groupByKey</td>
<td>ProvenancePairRDD</td>
</tr>
<tr>
<td>key.(String,String)</td>
<td>Value: Float</td>
</tr>
<tr>
<td>mapValues</td>
<td>ProvenancePairRDD</td>
</tr>
<tr>
<td>key.(String,String)</td>
<td>Value: List[TaintedFloat]</td>
</tr>
<tr>
<td>groupByKey</td>
<td>ProvenancePairRDD</td>
</tr>
<tr>
<td>key.(String,String)</td>
<td>Value: Float</td>
</tr>
<tr>
<td>mapValues</td>
<td>ProvenancePairRDD</td>
</tr>
<tr>
<td>key.(String,String)</td>
<td>Value: List[TaintedFloat]</td>
</tr>
</tbody>
</table>

Figure 3: Using textFileWithTaint, FLOWDEBUG automatically transforms the application DAG. ProvenanceRDD enables transformation-level provenance and influence-function capability, while tainted primitive types enable UDF-level taint propagation.

Figure 4: Running example 2 identifies, for each state in the US, the variance of snowfall reading for each day of any year and for any particular year. Red rectangle highlights the changes to enable influence-based provenance.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>val log = &quot;s3n://xcr:wYws/logs/weather.log&quot;</td>
</tr>
<tr>
<td>3</td>
<td>val split = input.flatMap(x: TaintedString =&gt;</td>
</tr>
<tr>
<td>4</td>
<td>.aggregateByKey((0.0, 0.0, 0)) {</td>
</tr>
<tr>
<td>5</td>
<td>case ((sum, sum_sq, count), next) =&gt;</td>
</tr>
<tr>
<td>6</td>
<td>(sum + next, sum_sq + next * next, count + 1),</td>
</tr>
<tr>
<td>7</td>
<td>case ((sum1, sum_sq1, count1),</td>
</tr>
<tr>
<td>8</td>
<td>(sum2, sum_sq2, count2)) =&gt;</td>
</tr>
<tr>
<td>9</td>
<td>(sum1 + sum2, sum_sq1 + sum_sq2,</td>
</tr>
<tr>
<td>10</td>
<td>count1 + count2),</td>
</tr>
<tr>
<td>11</td>
<td>// Optional argument to enable taint propagation</td>
</tr>
<tr>
<td>12</td>
<td>enableTaintPropagation = Some(false),</td>
</tr>
<tr>
<td>13</td>
<td>// Optional influence function</td>
</tr>
<tr>
<td>14</td>
<td>influenceTrackerCtr = Some(</td>
</tr>
<tr>
<td>15</td>
<td>StreamingOutlierInfluenceTracker(</td>
</tr>
<tr>
<td>16</td>
<td>zscoreThreshold=0.96)</td>
</tr>
<tr>
<td>17</td>
<td>) =&gt; StreamingOutlierInfluenceTracker(</td>
</tr>
<tr>
<td>18</td>
<td>enableTaintPropagation = Some(false),</td>
</tr>
<tr>
<td>19</td>
<td>influenceTrackerCtr = Some(</td>
</tr>
<tr>
<td>20</td>
<td>StreamingOutlierInfluenceTracker(</td>
</tr>
<tr>
<td>21</td>
<td>zscoreThreshold=0.96)</td>
</tr>
<tr>
<td>22</td>
<td>)</td>
</tr>
<tr>
<td>23</td>
<td>).mapValues</td>
</tr>
<tr>
<td>24</td>
<td>case (sum, sum2, count) =&gt;</td>
</tr>
<tr>
<td>25</td>
<td>((count<em>sum2) - (sum</em>sum))/(count*(count-1))</td>
</tr>
<tr>
<td>26</td>
<td>deltaSnow.saveAsTextFile(&quot;hdfs://s3n://xcr:wYws/logs/weather&quot;.score&quot;)</td>
</tr>
</tbody>
</table>
2.2 Running Example 2
Consider another Apache Spark program shown in Figure 4. For each state of the US, this program finds the statistical variance of snowfall readings for each day in a calendar year and for each year. Similar to Example 1 in Figure 1b lines 4-19, the first transformation flatMap projects each input record into two records (truncated in Figure 4 line 4-6, corresponding to the same operation in Figure 1b): (state, mm/dd), and its snowfall measurement (state, yyyy), and its snowfall measurement. To find the variance of snowfall readings, we use aggregateByKey and mapValue operators to collectively group the incoming data based on the key (i.e., by state and day and by state and year) and incrementally compute the variance as we encounter new data records in each group. In vanilla Apache Spark, the API of aggregateByKey has two input parameters i.e., a UDF that combines a single value with partially aggregated values and another UDF that combines two sets of partially aggregated values. Further details of the API usage of aggregateByKey can be found elsewhere [1]. In Example 2, aggregateByKey returns a sum of squares, a sum of sum, and a count for each key group, which are used by mapValues to compute variance.

After inspecting the results of Example 2 on the entire data, we find that some output records have significantly high variance than the rest of the outputs such as [AK, 9/02, 1766085] as mentioned earlier, common debugging practices such as simple scans on the entire input to search for extreme snowfall values are insufficient.

Imprecision of Data Provenance. Because Example 2 calculates statistical variance for each group, data provenance techniques consider all input records within a group are responsible for generating an output, as they do not distinguish the degree of influence of each input record on the aggregated output. Thus all inputs records that map to a faulty key-group are returned and the size could still be in millions of records (in this case 6,063,000 records), which is infeasible to inspect manually. For debugging purposes, a user may want to see the input, within the isolated group, that has the biggest influence on the final variance value. For example, in a set of numbers (1, 2, 3, 4, 4, 4, 99), a number 4 is closer to the average 15 and has less influence on the variance than the number 99, which is the farthest away from the average.

Inefficiency of Search-based Debugging. The limitation of search-based debugging approaches such as BigSift [18] and DD [41] is that they require a test oracle function that satisfies the property of unambiguity—i.e., the test failure should be caused by only one segment, when the input is split into two segments. For Figure 4, the final statistical variance output of greater than 1,750,000 is marked as incorrect, as it is slightly higher than the other half. BigSift applies DD on the backward trace of the faulty output and isolates the following two input records as faulty:

\[29749, 9/2/1976, 3352\text{mm} \]
\[29749, 9/2/1933, 39\text{nm} \]

Although the two input records fail the test function, they are completely valid inputs and should not considered as faulty. This false positive is due to the violation of the unambiguity assumption. During the fault isolation process, DD restricts its search on the first half of the input, assuming that none of the second half set leads to a test failure. However, in our case, there are multiple input subsets that could cause a test failure, and only one of those subsets contains the real faulty input. Therefore, DD either returns correct records as faulty or does not return anything at all.

Debugging Example 2 with FlowDEBUG. Similar to Example 1, a user can replace SparkContext with FlowDebugContext. This change automatically replaces all the succeeding RDDs with ProvenanceRDDs. As a result, split at line 4 of Figure 4 becomes ProvenanceRDD which uses the refactored version of all aggregation operators APIs provided by FlowDEBUG (e.g., reduce or aggregateByKey). These APIs include optional parameters: (1) enableTaintPropagation, a toggle to enable or disable taint propagation and (2) influenceTrackerCtr, an influence function to rank input records based on their impact on the final aggregated value. The user only need to make the edits shown in the red rectangle to enable influence-based data provenance (Figure 4), and the rest of taint tracking is done fully automatically by FlowDEBUG. A user may select one of many predefined influence functions described in Section 3.3 or can provide their own custom influence function to define selectivity and priority for debugging aggregation logic. Lines 8-22 of Figure 4 show the invocation of aggregateByKey that takes in an influence function StreamingOutlierInfluenceTracker to prioritize records with extreme snowfall readings. The guidelines of writing an influence function is presented in Section 3.3. Based on this influence function, FlowDEBUG keeps the input records with the highest influence only and propagates their provenance to the next operation i.e., mapValues. Finally, it returns a reference pointing to an input record that has the largest impact on the suspicious variance output.

77202, 7/12/1933, 90 in

3 APPROACH
FlowDEBUG is implemented as an extension library on top of Apache Spark’s RDD APIs. After a user has imported FlowDEBUG APIs, provenance tracking is automatically enabled and supported in three steps. First, FlowDEBUG assigns a unique provenance ID to each record in any initial
source RDDs. Second, it runs the program and propagates a set of provenance IDs alongside each record in the form of data-provenance pairs. As the provenance for any given data record may vary greatly depending on application semantics, it utilizes an efficient RoaringBitmap [29] for storing the provenance ID sets. Finally, when a user queries for input records responsible for a given output, it retrieves the provenance IDs for each of the inputs and joins them against the source RDDs from the first step to produce the final subset of input records. Figure 5 shows the propagation of provenance at both operator-level and UDF-level and how influence function refines provenance tracking at aggregation operators.

3.1 Transformation Level Provenance

ProvenanceRDD API mirrors Spark’s RDD API and enables developers to easily apply FLOWDEBUG to their existing Spark applications with minimal changes. An example edit to enable taint tracking is shown in Figure 1b. As provenance is paired with each intermediate or output data record, provenance propagation can be broken down into the following:

- For one-to-one dependencies, provenance propagation requires copying the provenance of the input record to the resulting output record. Such dependencies stem from RDD operations such as map and filter.

- For many-to-one mappings, the provenance of all input records is unioned into a single instance. Examples of many-to-one mappings include combineByKey and reduceByKey.

- For one-to-many mappings created by flatMap, FLOWDEBUG considers them as multiple dependencies sharing the same source(s).

FLOWDEBUG enables higher precision provenance tracking than operator level provenance by propagating taints within UDFs using tainted data types (Section 3.2), and by leveraging influence functions (Section 3.3).

3.2 UDF-Aware Tainting

FLOWDEBUG enables UDF-aware taint tracking. This mode rewrites the given program to automatically convert the supported data types into corresponding, tainting-enabled data types that store both the original data type object along with a set of provenance tags. These tainted data types mirror the APIs of their original data types, but propagate provenance information through UDFs to produce new, refined taints.

For example, in Figure 6, the UDF on the left takes a collection of records and their corresponding taints as inputs and selects only numeric string e.g., "2.1". In such cases, FLOWDEBUG performs control-flow aware tainting and removes the taints of filtered-out records i.e., taint 3343 and 3344 for records "$A/N$". Similarly, the UDF on the right takes in a collection of records and sums up values that are less than 10.0. FLOWDEBUG’s data-flow aware tainting captures such interactions and merges the provenances of only the records less than ten i.e., taint 3341 and 3345 for records "$A/2.1$" and "$A/6.9$" respectively. Since traditional provenance techniques do not understand the UDF semantics, they map the output record "$A/6.9$" to all elements in the input collection with taint [3341,3342,3343,3344,3345,3346].

Tainted Data Types. FLOWDEBUG individually retains provenance for each tainted data type. When multiple taints interact with each other through the use of binary or tertiary operators (e.g., addition of two numbers), the two sets of provenance tags are merged to produce the output taint set. FLOWDEBUG currently supports all common Scala data types and operations, broken down into numeric and string types.

Numeric Taint Types. FLOWDEBUG provides tainted data types for Scala’s Int, Long, Double, and Float numeric types. Standard operations such as arithmetic and conversion to other tainted data types are extended to produce corresponding tainted data objects. Many common numerical operations are not explicitly part of Scala’s Numeric APIs. To support operations such as 

Math.max, FLOWDEBUG provides an equivalent library of predefined numerical operations for its tainted...
3.3 Influence Function Based Provenance

The dataflow operator-level provenance described in Section 3.1 suffers from the same issue of over-approximation that other techniques have [15, 23, 30]. This shortcoming inherently stems from the black box treatment of UDFs passed an argument to aggregation operators such as reduceByKey. For example, in Figure 9, aggregateByKey’s UDF computes statistical variance. Although all input records contribute towards computing variance, input numbers with anomalous values have greater influence than other. Traditional data provenance techniques cannot detect such interaction and map all input records to the final aggregated value.

FLOWDEBUG provides additional options in the ProvenanceRDD API to selectively choose which input records have greater influence on the outcome of aggregation. This extension mirrors Spark’s combineByKey API by providing init, mergeValue, and mergeFunction methods which allow customization for how provenance is filtered and prioritized for aggregation functions:

- init(value, provenance): Initialize an influence function object with the provided data value and provenance.
- mergeValue(value, provenance): Add another value and its provenance to an already initialized influence function, updating the provenance if necessary.
- mergeFunction(influenceFunction): Merge an existing influence function (which may already be initialized and updated with values) into the current instance.
- finalize(): Compute any final postprocessing step and return a single provenance object for all values observed by the influence function.

Developers can define their own custom influence functions or use predefined, parameterized influence function implementations provided by FLOWDEBUG as a library, described in Table 1. The influence function API is intended for flexibility and its efficacy ultimately depends on the users’ choice. Figure 8 presents an example implementation of the influence function API and the predefined Custom Filter influence function. StreamingOutlier provides a general implementation as an effective starting point. Although how automatic construct influence functions for arbitrary aggregation UDFs remains an open challenge, FLOWDEBUG’s predefined influence functions should address the majority of use cases. As an example, suppose a developer is trying to debug a program which computes a per-key average that yields an abnormally high value. She may choose to use a TopN influence function and retain only the top ten values’ provenance within each key. Using this influence function, FLOWDEBUG can then reduce the number of inputs traced to a more manageable subset for developer inspection.

```scala
class case class TaintedString(value: String, p: Provenance) extends TaintedAny(value, p) {
  def toInt: TaintedInt =
    TaintedInt(value.toInt, getProvenance())
  def equals(obj: TaintedString): Boolean =
    value.equals(obj.value)
}
```

```scala
class FilterInfluenceFunction[T](filterFn: T => Boolean) extends InfluenceFunction[T] {
  private val values = ArrayBuffer[T]{}
  def addIfFiltered(value: T, prov: Provenance) {
    if (filterFn(value)) values += prov
  } 
  override def init(value: T, prov: Provenance) =
    addIfFiltered(value, prov)
  override def mergeValue(value: T, prov: Provenance) =
    addIfFiltered(value, prov)
  override def mergeFunction(other: InfluenceFunction[T]) =
    other match {
      case o: FilterInfluenceFunction[T] =>
        this.values ++= o.values
      case _ =>
        this
    }
  override def finalize(): Provenance = {
    values.reduce((case a,b) => a.union(b))
  }
}
```

Figure 7: TaintedString intercepts String’s method calls to propagate the provenance by implementing Scala.String methods.

Figure 8: The implementation of the predefined Custom Filter influence function, which uses a provided boolean function to evaluate which values’ provenance to retain.

- `mergeValue(value, provenance)`: Add another value and its provenance to an already initialized influence function, updating the provenance if necessary.
- `mergeFunction(influenceFunction)`: Merge an existing influence function (which may already be initialized and updated with values) into the current instance.
- `finalize()`: Compute any final postprocessing step and return a single provenance object for all values observed by the influence function.
Due to the stateless nature of batch processing in Apache Spark, FLOWDEBUG’s results may not generalize to stateful approaches. Compared to Titian, FLOWDEBUG improves debugging precision by 15.0 to 99.9 percentage points by using influence functions and taint analysis in tandem to discard irrelevant inputs. Despite this improvement in precision and trace size reduction, FLOWDEBUG achieves the same 100% recall as Titian. Compared to BigSift, FLOWDEBUG’s recall is 96.8 to 99.3 percentage points higher.

Finally, FLOWDEBUG’s combined instrumentation and tracing time is 12X - 51X faster than Titian and 500X - 1000X faster than BigSift because FLOWDEBUG actively propagates finer-grained provenance information, thus reducing its backwards tracing time. Additionally, FLOWDEBUG’s debugging time is faster than BigSift because it does not require multiple re-executions to improve its tracing precision. Compared

Table 1: Influence function implementations provided by FLOWDEBUG.

<table>
<thead>
<tr>
<th>Influence function</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>None</td>
<td>Returns all provenance IDs. This is the default behavior used in transformation level provenance, when no additional UDF information is available.</td>
</tr>
<tr>
<td>TopN/BottomN</td>
<td>N (integer)</td>
<td>Retain provenance of the N largest/smallest values.</td>
</tr>
<tr>
<td>Custom Filter</td>
<td>FilterFn (boolean function)</td>
<td>Use a provided Scala boolean filter function (FilterFn) to evaluate whether or not to retain provenance for consumed values.</td>
</tr>
<tr>
<td>StreamingOutlier</td>
<td>Z (integer), BufferSize (integer)</td>
<td>Retain values that are considered outliers as defined by Z standard deviations from the (streaming) mean, evaluated after BufferSize values are consumed. The default values are Z=3, BufferSize=1000.</td>
</tr>
<tr>
<td>Union</td>
<td>InfluenceFunctions (1+ Influence Functions)</td>
<td>Apply each provided influence function and calculate the union of provenance across all functions.</td>
</tr>
</tbody>
</table>

Figure 9: Comparison of operator-based provenance (blue) vs. influence-function based provenance (red). The aggregation logic computes the variance of a collection of input numbers.

The running time is broken into two parts: (1) the instrumented running time shown in Figure 10, as all three tools capture and store provenance tags by executing an instrumented program, and (2) the debugging time shown in Figure 11, as all three tools perform backward tracing for each faulty output to identify a set of relevant inputs records. Note that Titian and BigSift are not required in FLOWDEBUG’s workflow, and they are only used as baselines for comparisons.

The results highlight a few major advantages of FLOWDEBUG over existing data provenance (Titian) and search-based debugging (BigSift) approaches. Compared to Titian, FLOWDEBUG improves debugging precision by 15.0 to 99.9 percentage points by using influence functions and taint analysis in tandem to discard irrelevant inputs. Despite this improvement in precision and trace size reduction, FLOWDEBUG achieves the same 100% recall as Titian. Compared to BigSift, FLOWDEBUG’s recall is 96.8 to 99.3 percentage points higher.

4 EVALUATION

We investigate six programs and compare FLOWDEBUG to Titian, BigSift in precision, recall, and the number of inputs that each tool traces from the same set of faulty output records. We also compare debugging (tracing) times for all three tools, as well as application execution times for the same tools plus Apache Spark. Our programs and datasets are adopted from prior work [18]. We use the open source versions of Titian and BigSift with no additional modifications, as well as Apache Spark 2.2.0. Each program is evaluated on a single machine running macOS 10.15.3 with 16GB RAM, a 2.6GHz 6-core Intel Core i7 processor, and 512GB flash storage.

Table 2, Figure 10, and Figure 11 summarize the results. The running time is broken into two parts: (1) the instrumented running time shown in Figure 10, as all three tools capture and store provenance tags by executing an instrumented program, and (2) the debugging time shown in Figure 11, as all three tools perform backward tracing for each faulty output to identify a set of relevant inputs records. Note that Titian and BigSift are not required in FLOWDEBUG’s workflow, and they are only used as baselines for comparisons.

The results highlight a few major advantages of FLOWDEBUG over existing data provenance (Titian) and search-based debugging (BigSift) approaches. Compared to Titian, FLOWDEBUG improves debugging precision by 15.0 to 99.9 percentage points by using influence functions and taint analysis in tandem to discard irrelevant inputs. Despite this improvement in precision and trace size reduction, FLOWDEBUG achieves the same 100% recall as Titian. Compared to BigSift, FLOWDEBUG’s recall is 96.8 to 99.3 percentage points higher.

Finally, FLOWDEBUG’s combined instrumentation and tracing time is 12X - 51X faster than Titian and 500X - 1000X faster than BigSift because FLOWDEBUG actively propagates finer-grained provenance information, thus reducing its backwards tracing time. Additionally, FLOWDEBUG’s debugging time is faster than BigSift because it does not require multiple re-executions to improve its tracing precision. Compared
### Table 2: Debugging accuracy results for Titian, BigSift, and FlowDebug.

<table>
<thead>
<tr>
<th>Program</th>
<th>Faulty Outputs</th>
<th>Trace Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Department GPA</td>
<td>UDFAware Tainting</td>
<td>6,063,000</td>
</tr>
<tr>
<td>Course Avg</td>
<td>UDFAware Tainting, Top2+Bottom2</td>
<td>6,247,562</td>
</tr>
<tr>
<td>Student Info</td>
<td>StreamingOutlier(&gt;=3)</td>
<td>713,760</td>
</tr>
<tr>
<td>Commute Type</td>
<td>TopN(N=1000)</td>
<td>9,545,636</td>
</tr>
</tbody>
</table>

### Figure 10: The instrumented running time of FlowDebug, Titian, BigSift, and Spark.

The Weather Analysis program runs on a dataset of 42 million rows consisting of comma-separated strings of the form "zip code, day/month/year, snowfall amount (mm or ft)". It parses each string and calculates the largest delta, in millimeters, between the minimum and maximum snowfall readings for each year as well as each day and month. After running this program, we find 76 output records that are abnormally high and each contains a delta of over 6000 millimeters.

We first attempt to debug this issue using Titian by initiating a backward trace on these 76 faulty outputs. Titian returns 6,063,000 records, which correspond to over 14% of the entire input. Such a large number of records is infeasible for a developer to inspect. Because the UDF passed to the aggregation operator uses only min and max, the delta being computed for each key group should correspond to only two records per group. However, Titian is unable to analyze such UDF semantics and instead over-approximates the provenance of each output record to all inputs with the same key.

**FlowDebug** precisely accounts for these UDF semantics by leveraging UDFAware tainting which rewrites the application to use tainted data types. As a result, it returns a much more manageable set of 112 input records. A quick visual inspection reveals that 40 of these inputs have their snowfall measurements listed in inches, which are not considered by the UDF. The program thus converts these records to millimeters as if they were in feet, which is the root cause for the unusually high deltas in the faulty outputs.

In terms of instrumentation overhead, FlowDebug takes 86 seconds while Titian takes 57 seconds, as shown in Figure 10. FlowDebug’s tracing time is significantly faster at just under 3 seconds, compared to the 67 seconds taken by Titian, as shown in Figure 11.

Another alternative debugging approach may have been to use BigSift to isolate a minimal fault-inducing subset. BigSift yields exactly two inputs, one of which is a true fault containing an inch measurement. However, the small size of this result set makes it difficult for developers to diagnose the underlying root cause as it may be difficult to generalize results from a single fault. The debugging time for BigSift is unreasonably expensive on the dataset of 42 million records, as it requires 41 reruns of the program with different inputs and takes over 400 times longer than FlowDebug (Figure 11).

### 4.2 Airport Transit Analysis

The Airport Transit Analysis program runs on a dataset of 36 million rows of the form "date, passengerID,
arrival, departure, airport”. It parses each string and calculates the sum of layover times for each pair of an airport location and a departure hour as follows:

```scala
1 // number of minutes elapsed between two times
2 def getDiff(arr: String, dept: String): Int = {
3   val arr_min = arr.split(":")(0).toInt * 60 +
4     arr.split(":")(1).toInt
5   val dep_min = dept.split(":")(0).toInt * 60 +
6     dept.split(":")(1).toInt
7   if (dep_min - arr_min < 0) {
8     return dep_min - arr_min + 24*60
9   }
10  return dep_min - arr_min
11 }
12 val tokens = s.split(" ,")
13 val airport = tokens(4)
14 val dept_hr = tokens(3).split(" - ")(0)
15 val diff = getDiff(tokens(2), tokens(3))
16 (airport, dept_hr, diff)
17 }
18 val result = input.reduceByKey(_.+).collect()
```

Unfortunately, after running this program, 33 of 384 produced outputs have a negative value that should not be possible.

To understand why, we use Titian to trace these faulty outputs. Titian returns 773,760 input records, the vast majority of which do not have any noticeable issues. Without any specific insight as to why the faulty sums are negative, we enable FLOWDEBUG with the StreamingOutlier influence function using the default parameter of z=3 standard deviations. FLOWDEBUG reports a significantly smaller set of 34 input records, all of which have departure hours greater than the expected [0.24] range. As a result, the program’s calculation of layover duration ends up producing a large negative value for these trips, which is the root cause of faulty outputs.

FLOWDEBUG precisely identifies all 34 faulty input records with over 99.9 percentage points more precision than Titian and a smaller result size that developers can more easily inspect. Additionally, FLOWDEBUG produces these results significantly faster; Figure 10 shows that Titian’s instrumented run takes 100 seconds, which is 5 times more than FLOWDEBUG’s. Furthermore, FLOWDEBUG’s backward tracing takes 2 seconds compared to 42 seconds by Titian, as shown in Figure 11.

For additional comparison, BigSift takes 550 times as long as FLOWDEBUG (Figure 11) to trace a single faulty record that represents in a decrease of 97.1 percentage points in recall. The root cause of the error (out-of-range departure hours) is also not immediately clear from this single record without careful code inspection or additional data points to generalize from.

4.3 Departmental GPA Statistics Analysis
The Departmental GPA Statistics program operates on 25 million rows consisting of “studentID, courseNumber, grade”. It parses each string entry and computes the GPA bucket for each grade on a 4.0 scale. Next, the program computes the average GPA per course number. Finally, it computes the mean and variance of course GPAs in each department.

When we run the program, we observe the following output:

- CS, (2.728, 0.017)
- Physics, (2.713, 3.339E-4)
- MATH, (2.715, 3.594E-4)
- EE, (2.715, 3.338E-4)
- STATS, (2.712, 3.711E-4)

Strangely, the CS department appears to have an unusually higher mean and variance than the other departments. There are approximately 5 million rows belonging to the CS department across about a thousand different course offerings, and a quick visual sample of these rows does not immediately highlight any potential fault cause due to the variety of records and complex aggregation logic in the program.

Instead, we opt to use FLOWDEBUG’s influence function mode and its StreamingOutlier influence function with the default parameter of z=3 standard deviations. We rerun our application with this influence function and trace the CS department record, which yields 50,370 records. While still a large number, a brief visual inspection quickly reveals an abnormal trend where all the records originate from only two courses: CS9 and CS11. Upon computing the course GPA for these two courses, we find that it is significantly greater than most other courses—whereas most courses hover around a GPA average of 2.7, these two courses have an unusually high GPA average of 4.0. As a result, these two courses skew the CS department mean and variance to be higher than those of other departments. For the Departmental GPA Statistics program, neither Titian nor BigSift produce any input traces. BigSift is not applicable to this program due to its unambiguity requirement for a test oracle function.

4.4 Course Grade Averages Analysis
The Course Grade Averages program operates on a modified version of the Departmental GPA Statistics dataset, consisting of approximately 50 million rows in the form "studentID, courseNumber, grade". First, the program parses each string record and additionally extracts a department based on the course number. Next, it computes the average grade for each course, as shown in Figure 12. Using these course averages, the program then computes averages for the top-5 and bottom-5 course averages within each department (see Figure 12). After execution, we observe the following output:

- (CS, (81.67, 82.31))
- (STATS, (81.69, 82.31))
- (MATH, (81.66, 82.37))
- (Physics, (78.00, 82.35))
- (EE, (81.60, 82.33))
Note that the Physics department appears to have an abnormal bottom-5 average, 78, compared to the other departments. However, there are approximately 10 million rows belonging to the Physics department so manual inspection is infeasible.

Because the program uses two aggregation steps, we enable both influence function and UDF-aware tainting options in FLOWDEBUG. For the initial aggregation of course averages (I in Figure 12), we use a Top-2 and Bottom-2 influence function that identifies the two highest and the two lowest performing students in each course. We address the second aggregation (II in Figure 12) by enabling UDF-aware tainting to track the concise set of courses whose grade averages are either in the top 5 or bottom 5 within their department.

For the abnormal bottom-5 average of 78 in the Physics department, FLOWDEBUG uses UDF-aware tainting to trace the exact 5 Physics courses with the lowest course averages. Each course is then traced back to 4 input records (i.e., the two highest and the two lowest performing students) via the influence function. In other words, FLOWDEBUG returns the top 2 and bottom 2 student grades for each of the 5 Physics courses with the lowest course averages.

In practice, 2 of the 5 traced Physics courses have abnormally low averages of 72.36 and 72.65, owing to 24 students with "0" grades. FLOWDEBUG’s influence function identifies 4 of those 24 students, i.e., the two lowest-performing students in each course. The rest of the traced input includes, from each course, the two best performing students and the two lowest-performing students with non-zero grades. Therefore, FLOWDEBUG achieves a precision of 20% and a recall of approximately 16.67%. Note that these measurements are innately tied to the influence function and the program semantics. For example, UDF-aware tainting alone can achieve at best 40% precision in this scenario since it must trace 5 courses while only 2 contain faults.

The Course Grade Averages program also demonstrates the combined benefits of taint analysis and influence functions, as opposed to using each in isolation. If only UDF-aware tainting is enabled for this program, FLOWDEBUG cannot reduce the lineage size across the first aggregation (I in Figure 12). Thus, it returns all the entries that belong to the bottom 5 Physics courses. With 153,320 such entries, this approach results in a recall of 100% but a precision of 0.016%. When enabling influence function only, FLOWDEBUG identifies the 4 (Top-2 and Bottom-2) most influential input records for every Physics course, instead of finding the bottom 5 courses in Physics. With 200 courses in the department, 800 records are traced and the approach results in a precision and recall of 0.5% and 16.67%, respectively.

The public release of Titian was unable to support the Course Grade Averages program due to its incompatibility with multiple aggregation stages. It runs out of memory on the specified dataset and returns an empty backwards trace for a smaller dataset, similar to the behavior observed in the Departmental GPA Statistics program. As BigSift directly depends on Titian, it also does not support this program.

### 4.5 Student Info Analysis

The Student Info Analysis program parses 25 million rows of data consisting of "studentId, major, gender, year, age" to compute an average age for each of the four typical college years. However, there appears to be a bug as the average age for the "Junior" group is 265 years old, much higher than the typical human lifespan.

To debug why, we use Titian to trace the faulty "Junior" output record only to find that it returns a subset of over 6.2 million input records. A quick visual sample does not reveal any glaring bug or commonalities among the records other than that they all belong to "Junior" students. Instead, we use FLOWDEBUG to identify a more precise input trace.

When using FLOWDEBUG’s StreamingOutlier influence function with the default parameter of $z=5$ standard deviations, FLOWDEBUG identifies a much smaller set of 31 input records. Inspection of these reveals that the student ID and age values are swapped, resulting in impossible ages such as "92611257" which drastically increase the overall average for the "Junior" key group.

FLOWDEBUG produces an input set that is both smaller and over 99.9 percentage points more precise than Titian. Additionally, FLOWDEBUG’s instrumented run takes 8 seconds, 8 times less than Titian’s, while its input trace takes 2 seconds compared to Titian’s 23 seconds. Overall, FLOWDEBUG finds fault-inducing inputs in approximately 8% of the original job processing time. Compared to BigSift, which reports a single faulty record after 31 program re-executions, FLOWDEBUG is over 500 times faster while providing higher recall and equivalent precision.

### 4.6 Commute Type Analysis

The Commute Type Analysis program begins with parsing 25 million rows of comma-separated values with the schema "zipCodeStart, zipCodeEnd, distanceTraveled, timeElapsed". Each record is grouped into one of three commute types—car, public transportation, and bicycle—according to its speed.
as calculated by distance over time in miles per hour. After computing the commute type and speed of each record, the average speed within each commute type is calculated by computing the sum and count within each group.

```scala
val trips = inputs.map { s: String => 
  val cols = s.split(',')
  val distance = cols(3).toInt
  val time = cols(4).toInt
  val speed = distance / time
  if (speed > 40) {
    if (speed > 40) { // car
      /*car", speed*/
    } else if (speed > 15) { // bicycle
      /*bicycle", speed*/
    } else { // public transportation
      /*public transportation", speed*/
    }
  }
  (distance, count)
}.mapValues {
  (sum, count) => (sum + next, count + 1)
}.collect
```

When we run the Commute Type Analysis program, we observe the following output:

```
car, 50.88
public transportation, 27.99
bicycle, 11.88
```

The large gap between public transportation speeds and car speeds is immediately concerning, as 50+ miles per hour is typically in the domain of highway speeds rather than daily commutes which typically include surface streets and traffic lights. To investigate why the average car speed is so high, we use Titian to conduct a backwards trace, Titian identifies approximately 9.5 million input records, which amounts to over one third of the entire input dataset. Due to the sheer size of the trace, it is difficult to comprehensively analyze the input records for any patterns that may cause the abnormally high average speed.

Instead, we choose to use FLOWDEBUG to reduce the size of the input trace. Since we know that the average speed is unexpectedly high, we configure FLOWDEBUG to use the TopN influence function with an initial parameter of n=1000 to trace the "car" output record. FLOWDEBUG returns 1000 input records, of which 150 records have impossibly high speeds of 500+ miles per hour. In fact, these records are airplane trips not accounted for.

FLOWDEBUG’s results are 15.0 percentage points more precise than Titian’s. Additionally, FLOWDEBUG’s instrumentation time (9 seconds) is much faster than Titian’s (57 seconds). A similar trend is shown for tracing fault-inducing inputs, where FLOWDEBUG takes under 2 seconds to isolate the faulty inputs while Titian takes 73 seconds. Note that our initial parameter choice of 1000 for our TopN influence function is an overestimate—larger values would increase the size of the input trace and processing time, while smaller values would have the opposite effect and might not capture all the faults present in the input. On the dataset of 25 million trips, BigSift pinpoints a single faulty record after 27 re-runs. However, this process takes over 1,100 times longer than FLOWDEBUG (Figure 11) and yields only a single input out of the 150 faulty inputs.

5 RELATED WORK

Data Provenance. Data provenance has been an active area of research in databases that can help explain how a certain query output is related to input data [15]. Data provenance has been successfully applied both in scientific workflows and databases [5, 6, 15, 21]. RAMP [23] and Newt [30] added data provenance support to DISC systems; both are capable of performing backward tracing of faults to failure-inducing inputs. Wu et al. design a new database engine, Smoke, that incorporates lineage logic within the dataflow operators and constructs a lineage query as the database query is being developed [36]. Ikeda et al. present provenance properties such as minimality and precision for individual transformation operators to support data provenance [22, 24]. Most of these data provenance approaches capture lineages at a coarse-grained, transformation operator level and do not analyze the semantics of the UDFs. On the other hand, FLOWDEBUG refines a set of inputs mapping to an output under investigation by incorporating dynamic tainting for UDFs. While we do not present a direct evaluation against RAMP and Newt, we do compare against Titian [25] which does including comparisons against RAMP and NEWT.

Taint Analysis. In software engineering, taint analysis is normally leveraged to perform security analysis [31, 34] and also used for debugging and testing [11, 28]. An example of such work is Penumbra [12] that automatically identifies the inputs related to a program failure by attaching fine-grained tags with program variables to track information flows. Program slicing is another technique that isolates statements or variables involved in generating a certain faulty output [4, 20, 39] using static and dynamic analysis. Chan et al. identify failure-inducing data by leveraging dynamic slicing and origin tracking [8]. DataTracker is another data provenance tool that slides in between the Linux Kernel and a Unix application binary to capture system-level provenance via dynamic tainting [38]. It intercepts systems calls such as open(), read(), and mmap2() to attach and analyze taint marks. Doing so on a large-scale distributed framework on top of JVM can be remarkably expensive as it would tag every system call, including those irrelevant to the DISC application. In general, applying these techniques to DISC applications...
can be costly, as they cannot distinguish the DISC framework code from UDF code. FLOWDEBUG applies three techniques in tandem—tainting within UDFs, coarse-grained data provenance, and influence-function based refinement—all of which are performed at the application level to overcome the limitation of over-approximating failure-relevant inputs.

**Search Based Debugging.** Delta debugging [43] has been used for a variety of applications to isolate a cause-effect chain or fault-inducing thread schedules [9, 13, 42]. As stated earlier, DD requires multiple re-executions of the program, which is expensive for DISC applications. One way to reduce the number of re-execution is to generate only valid configurations of inputs as implemented in HDD [33]. However, HDD assumes the input to be in a well defined hierarchical structure (e.g., XML, JSON), which only allows a very small number of valid input sub-configurations. This assumption does not hold true for DISC applications, as the input is usually unstructured or semi-structured. BigSift [18] combines data provenance [25] and DD with several systems optimizations. As shown in Section 4, BigSift still suffers from a large number of repetitive re-runs, which significantly increases the debugging time. Additionally, it is not easy for a user to write an appropriate test oracle function required by DD.

**Influence Function Based Debugging.** Prior work on the explainability of a database query uses the notion of influence to reason about an anomalous result. Such approaches eliminate groups of tuples from the input such that the remaining input, in isolation, does not lead to an anomalous result. The goal is to find the most influential groups of tuples, usually referred to as explanations [32, 37, 40]. Meliou et al. study causality in the database area and identify tuples responsible of answers (why) and non-answers (why-not) to queries [32] by introducing the degree of responsibility.

Scorpion [40] uses aggregation-specific partitioning strategies to construct a predicate that separates the most influential partition (subset of input). Here the notion of influence is that of a sensitivity analysis, where the generated predicate removes the input records which, if changed slightly, would lead to the biggest change in the outlier output. In other words, it finds the inputs records that are most sensitive to the outlier output instead of finding the most contributing inputs. Scorpion supports relational algebra in which the keys of group-by operator are explicitly mentioned in a structured data. However, in DISC applications, keys are extracted from unstructured data through arbitrarily complex UDFs. Scorpion also uses predefined partition strategies to decrease the search scope (similar to HDD [33]) and still requires repetitive executions of the SQL query.

With the goal of minimal provenance and output reproducibility in the context of differential dataflow, Chothia et al. [10] design custom rules for dataflow operators, *i.e.*, map, reduce, join to record record-level data *delta* at each operator for each iteration and for each increment of dataflow execution. Their approach in part resembles FLOWDEBUG’s StreamingOutlier influence function that captures influence over incremental computation. Applying their approach to a batch processing model requires partitioning the input and then capturing *delta* corresponding to every partition during incremental computation, making it drastically expensive both in terms of storage and runtime overhead.

Carbin et al. solve a similar problem of finding the influential (critical) regions in the input that have a higher impact on the output using fuzzed input, execution traces, and classification [7]. Compared to FLOWDEBUG, these approaches target structured data with relational or logical queries (e.g., datalog) to generate another counter-query to answer Why and Why not questions, but are not designed to handle the use of arbitrary, complex UDFs common in DISC applications.

**Debugging Big Data Analytics.** Gulzar et al. design a set of interactive debugging primitives such as simulated breakpoint and watchpoint features to perform breakpoint debugging of a DISC application running on cloud [19]. TagSniff introduces new debugging probes to monitor program states at runtime [14]. Upon inspection, a user can skip, resume, or perform a backward trace on a suspicious state. Other tools such as Arthur [16], Daphne [26], and Inspector Gadget [35] are also coarse grained and thus cannot isolate fault-inducing inputs precisely.

6 CONCLUSION

The data ingested by DISC applications is continuously evolving, often incomplete, and contains erroneous or invalid data that could cause failures or wrong outputs. Due to the terabyte scale of input data, developers find it challenging to distinguish faulty input records from billions of other records. Given a suspicious output, FLOWDEBUG identifies the precise record(s) that contribute the most towards generating the suspicious output. It introduces the notion of influence functions in data provenance to keep track of the most influential inputs and uses UDF-aware taint tracing to capture control flow and dataflow dependency. FLOWDEBUG improves precision by up to 99.9 percentage points and increases recall by up to 99.3 percentage points, compared to prior data provenance approaches. As a result, FLOWDEBUG can eliminate manual debugging effort from the user by producing a precise and accurate explanation for a failure or a wrong output.

**Acknowledgments.** We thank the anonymous reviewers for their comments and Malte Schwarzkopf for his guidance as a shepherd. The participants of this research are in part supported by NSF grants CCF-1764077, CCF-1527923, CCF-1723773, ONR grant N00014-18-1-2037, Intel CAPA grant, Samsung grant, Google PhD Fellowship, and Alexander von Humboldt Foundation.
REFERENCES


