BigSift: Automated Debugging of Big Data Analytics in Data-Intensive Scalable Computing

Muhammad Ali Gulzar
University of California, Los Angeles
USA
gulzar@cs.ucla.edu

Siman Wang
Hunan University
China
simanw@ucla.edu

Miryung Kim
University of California, Los Angeles
USA
miryung@cs.ucla.edu

ABSTRACT

Developing Big Data Analytics 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 pinpointing the root cause of errors and explaining the sources of anomalies. To address this problem, BigSift takes an Apache Spark program, a user-defined test oracle function, and a dataset as input and outputs a minimum set of input records that reproduces the same test failure by combining the insights from *delta debugging* with *data provenance*. The technical contribution of BigSift is the design of systems optimizations that bring automated debugging closer to a reality for data intensive scalable computing.

BigSift exposes an interactive web interface where a user can monitor a big data analytics job running remotely on the cloud, write a user-defined test oracle function, and then trigger the automated debugging process. BigSift also provides a set of predefined test oracle functions, which can be used for explaining common types of anomalies in big data analytics—for example, finding the origin of the output value that is more than k standard deviations away from the median. The demonstration video is available at https://youtu.be/jdBsCd61a1Q.

CCS CONCEPTS

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

KEYWORDS

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

ACM Reference Format:
Muhammad Ali Gulzar, Siman Wang, and Miryung Kim. 2018. BigSift: Automated Debugging of Big Data Analytics in Data-Intensive Scalable Computing. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236024.3264586

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264586

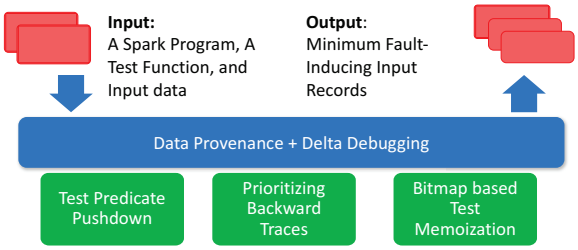


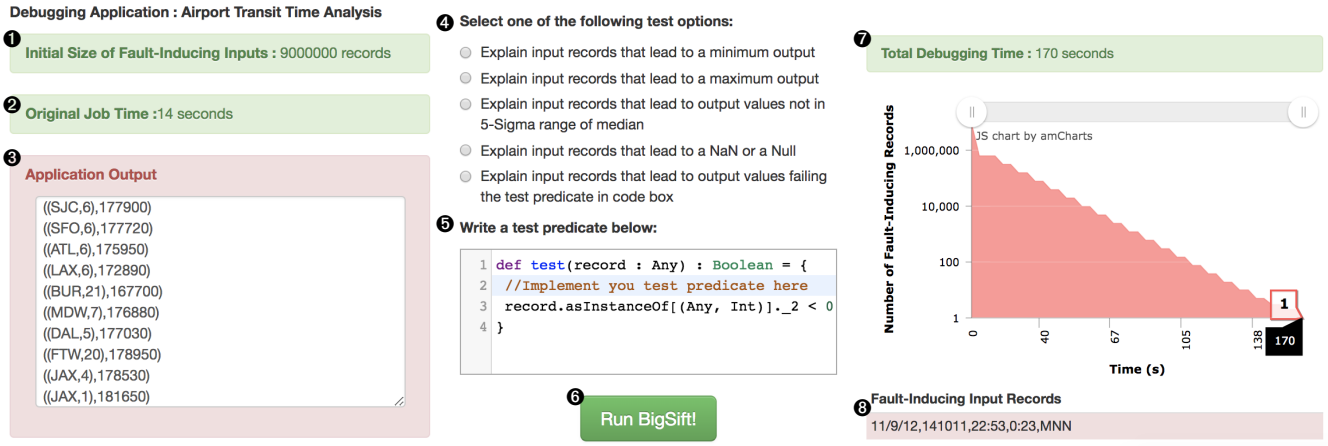
Figure 1: BigSift Overall Architecture

1 INTRODUCTION

Data-Intensive Scalable Computing (DISC) systems such as Google's MapReduce, Apache Spark, and Apache Hadoop enable processing massive data sets. Similar to other software development platforms, developers often deal with unclean data or make wrong (or incomplete) assumptions about the data. It is therefore crucial to equip these developers with toolkits that can better pinpoint the root cause of an error. Unfortunately, debugging big data analytics is currently an ad-hoc, time-consuming process. Data scientists typically write code that implements a data processing pipeline and test it on their local development workstation with a small sample data, downloaded from a TB-scale data warehouse. They cross fingers and hope that the program works in the expensive production cloud. When a job fails or they get results that end up being suspicious, data scientists must identify the source of the error, often by digging through post-mortem logs.

In such cases, the programmer (*e.g.* data scientist) may want to pinpoint the root cause of errors by investigating a subset of corresponding input records. One possible approach is to track *data provenance* (input output record mappings created in individual distributed worker nodes). However, according to our prior study [1], backward tracing based on *data provenance* finds an input subset in the order of millions, which is still too large for a developer to manually sift through. *Delta Debugging* (DD) is a well-known algorithm that re-executes the same program with different subsets of input records [10]. Applying the DD algorithm naively on big data analytics is not scalable because DD is a generic, black box procedure that does not consider the key-value mapping generated from individual dataflow operators. Therefore, DD cannot prune irrelevant input records easily by considering the semantics of dataflow operators.

The technical contribution of BigSift is two folds. First, it combines delta debugging with *data provenance*. Second, it implements three systems-level optimizations—(1) test predicate pushdown, (2) backward trace prioritization, and (3) bitmap-based memoization to be discussed in Section 2 in details—to improve debugging performance. Figure 1 shows the overall architecture of BigSift.



(a) Job completion time and output

(b) User and pre defined test function selection

(c) Area plot reports the real time information of the debugging process

Figure 2: BigSift's Web-based User Interface

Our evaluations show that BigSift improves the accuracy of fault localizability by several orders-of-magnitude (~10^3 to $10^7\times$) compared to Titian's [4] data provenance only. BigSift improves performance by up to 66× compared to using Delta Debugging alone [1]. For each faulty output, BigSift is able to localize fault-inducing data in less than 62% of the original job running time.

This tool demonstration paper builds on our prior work [1] and focuses on the tool features and corresponding implementation details of BigSift. BigSift is fully integrated with the current Apache Spark's web-based UI. A user can directly inspect raw output records, and write a test-oracle function on the fly or select from pre-defined test oracle functions. BigSift streams real time debugging progress information from the remote cluster to the user through an interactive area plot and presents the current set of fault-inducing input records in a table format. Our current implementation targets Apache Spark 2.1.1 with programs written in Scala and Java [9].

2 TECHNICAL APPROACH

The contribution of BigSift is to adapt *delta debugging* for big data analytics by designing new systems optimizations and by leveraging *data provenance* in tandem, which provides backward and forward tracing capabilities for Apache Spark [4]. The overview of our approach is described in Figure 1. Without such systems optimizations, delta debugging could take *hours* if not *days*. This is because the input dataset size is huge and thus an exhaustive, binary-search like algorithm such as delta debugging could take significant amount of time. In our evaluation, BigSift is up to 66 times faster than DD. Below we summarize three systems optimizations at a high level, and further details are described elsewhere [1].

2.1 Test Function Push Down.

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 in order to minimize network communication. Since delta debugging 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.

In Apache Spark, certain aggregation operators (*e.g.* `reduceByKey`) require a user to provide an *associative* and *commutative* function as an argument. BigSift implements a new optimization by pushing down a user-defined test function to partitions in the previous stage to test intermediate results. This optimization is enabled when (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. If this monotonicity property is not satisfied (which can be verified by testing final output), or none of the partitions fail the test function, BigSift rolls back to the default case of backward tracing the final faulty record.

2.2 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. Therefore, BigSift prioritizes the common input records leading to multiple outputs before applying DD. To check the eligibility for this optimization, BigSift explores a program DAG to find at least one 1-to-many or many-to-many operator such as `flatMap` and `join`.

In order to explore all the possible overlapping traces, BigSift overlaps the two smallest backward traces (let's say t_1 and t_2), to find the intersection, $t_1 \cap t_2$. If the test function evaluated on $t_1 \cap t_2$ 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 $t_2 - t_1$. 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 overlapped trace twice.

2.3 Bitmap Based Memoization of Test Results

DD is not capable of detecting redundant trials of the same input configuration and therefore may test the same input configuration multiple times. To avoid waste of computational resources, BigSift uses a *test results memoization* optimization. A naive memoization

```
class BigSift(sc:SparkContext, logFile:String){
  def runWithBigSift[T](
   sparkProgram : (RDD[String],Lineage[String]) =>
        RDD[T] ,   test : T => Boolean ) : Unit
  ... }
```

Figure 3: BigSift's API

strategy would require scanning of the content of an input configuration to check whether it was tested already; such content-based memoization would be time consuming and not scalable. BigSift instead leverages *bitmaps* to compactly encode the offsets of the input 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. 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 [5].

2.4 Implementation

To enable automated debugging of big data analytics applications, a user can instantiate `BigSift` class with `SparkContext` and input file path as input arguments, as shown in Figure 3. Internally, this class instantiates `LineageContext` that enables Titian's instrumentation for data provenance support. More details on the usage of Titian is described in our prior VLDB 2016 paper [4]. A user can then call `runWithBigSift` method with a test oracle function, and a `sparkProgram`—a directly acyclic graph (DAG) workflow that takes in an input Resilient Distributed Dataset (RDD–i.e., an abstraction of distributed collection) and returns the final RDD. BigSift is designed as an external Java library (jar) and can be deployed by importing the jar file in a Spark application running on a data-provenance enabled Spark distribution such as Titian [4]. BigSift's interactive UI is available on port 8989 on the Spark driver node. Figure 2 shows the web-based user interface. Once the job is completed, a user can examine the job execution time, raw output, etc. She can write her own custom test-oracle function or select from pre-defined test functions. BigSift also displays a set of input records that reproduce the same test failure. The area chart reports the real time debugging progress information. A user can click on the graph to see the size and samples of failure-inducing inputs.

3 DEMONSTRATION SCENARIO

Suppose Alice is a data scientist and she writes a big data application in Apache Spark to analyze a large scale dataset that contains passenger transit information in the US. Since the data is in the scale of terabytes, she takes a small sample of the dataset (say 10 MB) and builds a data processing pipeline using Spark in a local machine. Alice wants to find the total transit time for all passengers spending less than 45 minutes while in transit for each airport in

(a) Key-value output visualization (b) Data provenance visualization

Figure 4: BigSift's histogram visualization of key-value based output records

```
1  val countoftranist = sc.textFile(dataset).map{ s =>
2    val tokens = s.split(",")
3    val arrival_hr = tokens(2).split(":")(0)
4    val diff = getDiff(tokens(2) , tokens(3))
5    val airport = tokens(4)
6    ((airport, arrival_hr), diff)}
7  .filter{ v => v._2 < 45}
8  .reduceByKey(_+_)
9  .collect()
```

Figure 5: A Spark program written in Scala that finds the total layover time of all passengers spending less than 45 minutes per airport at each hour.

the US for each hour. A row in the dataset represents a passenger's transit information in the following format.

```
[date, passenger, arrival, departure, airport code]
        9/4/17 , 161413 , 6:52 , 8:22 , MNN
```

The program in Figure 5 first loads the dataset (line 1) and scans each row to retrieve a key-value pair. A key consists of the airport code and arrival hour of a passenger and the value is the transit time spent in minutes (departure time -arrival time) at the airport (line 2-6). Line 7 filters passengers with the transit time less than 45 minutes. Finally, the program sums up the transit times of all passengers per airport at each arrival hour (line 8).

After writing this application, Alice submits the job to the production cloud which results in the following output:

```
((SEA,7) , 175080)
((LAX,11) , 173460)
((MNN,23) , -27804120)
.....
```

She then realizes that some output records look suspicious. For example, the total transit time of `MNN` is `-27804120`, when she expects the total transit time to be a positive value. Alice wants to investigate what are the exact input records responsible for producing a negative value. This task is challenging because the large scale dataset is infeasible to inspect manually and there is no one-to-one mapping between input records and output records due to an aggregation step that applies user-defined functions.

Alice decides to use BigSift that takes her program, input data set, and a test oracle function as input and, eventually, returns the following culprit input record responsible for the suspicious negative output value.

```
11/9/12 , 141011 , 22:53 , 0:23 , MNN
```

The following describes BigSift demonstration step by step.

Step 1: Program Output Inspection. Figure 2 shows the landing page of BigSift. It shows the size of input dataset as the number of records, the job processing time, final output records in a text box. See ❶, ❷, and ❸ in Figure 2 respectively. To better visualize output records, BigSift provides interactive and dynamic visualization of key-value pairs using a histogram to make it easier for a user to identify anomalous records visually (Figure 4(a)). For example, Alice can mark any negative value as incorrect using a histogram and note down this threshold to construct a test function.

Step 2: Classifying Suspicious or Wrong Output Records by Defining a Test-Oracle Function. BigSift enables a user to write a test function—a predicate to be applied to each final output record to distinguish correct outputs from incorrect or anomalous outputs.

BigSift also enables user to choose from a list of pre-defined test predicate functions (Figure 2(b)-❹) to help explain the common types of anomalies in big data analytics: for example, (1) explain how a minimum output value is created, (2) explain how a maximum output value is created, (3) explain how the output value greater than k standard deviations from the median is created, etc. Once the selection is made from the radio buttons, a user can press the Run BigSift button (Figure 2(b)-❻). Internally, BigSift selects the corresponding pre-defined test function to initiate debugging.

Step 3: Visualization of Data Provenance. To help understand the propagation of fault-inducing intermediate input records across transformation steps, BigSift provides a pie chart based DAG visualization of the workflow (Figure 4(b)). Each node in this graph is represented as a pie chart where a red segment shows the ratio of fault-inducing intermediate records against the total number of records processed by that transformation. By viewing data ratio at each transformation, a user may get deeper insight.

Step 4: Automated DISC Debugging. When BigSift is invoked by the user, a realtime area chart appears on the UI. In Figure 2(c), Y-axis represents the number of fault-inducing input records isolated by BigSift in log scale and X-axis represents debugging time. As the time passes, BigSift streams debugging progress information from the cloud. A user can click on any part of the chart to view sample fault-inducing input records at the selected time. A mouse hover-over will show the number of fault-inducing input records. As soon as BigSift finds the minimum set of fault-inducing input records, BigSift reports the total debugging time through a push notification (green container in Figure 2(c)-❼).

4 RELATED WORK

Delta debugging (DD) is a well known technique for finding the minimal failure-inducing input [10] that requires multiple tests of the program, which alone, is not tractable for DISC system workloads. HDD tries to minimize DD tests by assuming that the input is in a well defined hierarchical structure which rarely holds [7]. RAMP [3] and Newt [6] add data provenance support to DISC systems. BigSift differs from these by leveraging DD and data provenance in tandem and by implementing unique systems optimizations to improve performance for DISC workloads. BigDebug is an interactive debugger for Spark [2] and it leaves to the developer to manually identify the root cause of errors. Data X-ray [8] extracts a set of features representing input data properties and summarizes the errors in a SQL table, but does not support automated debugging.

5 EVALUATION AND SUMMARY

We are in the early days of debugging big data analytics. This tool demonstration paper showcases BigSift, an automated debugging toolkit in the context of data-intensive scalable computing (DISC). Finding failure-inducing inputs is just the beginning. We see further opportunities for automated debugging of DISC applications, such as automated data cleaning and faulty code localization.

In our prior work [1], we evaluated BigSift on a 16-node cluster with 8 subject program where faults were injected in both input datasets or code. The datasets used in the evaluation ranges from few GB to 80GB. In comparison to using DD alone, BigSift reduced 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 on average 62% less than the original job running time per single faulty output.

ACKNOWLEDGMENT

We would like to thank Tyson Condie and Matteo Interlandi with their insights in the design of BigSift optimizations. Participants in this project are in part supported through AFRL grant FA8750-15-2-0075, NSF grants CCF-1764077, CCF-1527923, CCF-1460325, CCF-1723773, ONR grant N00014-18-1-2037, and gifts from Google and Huawei.

REFERENCES

[1] Muhammad Ali Gulzar, Matteo Interlandi, Xueyuan Han, Mingda Li, Tyson Condie, and Miryung Kim. 2017. Automated Debugging in Data-intensive Scalable Computing. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM, New York, NY, USA, 520–534. https://doi.org/10.1145/3127479.3131624

[2] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. 2016. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 784–795. https://doi.org/10.1145/2884781.2884813

[3] Robert Ikeda, Hyunjung Park, and Jennifer Widom. 2011. Provenance for generalized map and reduce workflows. In *In Proc. Conference on Innovative Data Systems Research (CIDR)*.

[4] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titian: Data Provenance Support in Spark. *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 216–227. https://doi.org/10.14778/2850583.2850595

[5] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. 2016. Consistently Faster and Smaller Compressed Bitmaps with Roaring. *Softw. Pract. Exper.* 46, 11 (Nov. 2016), 1547–1569. https://doi.org/10.1002/spe.2402

[6] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. 2013. Scalable lineage capture for debugging DISC analytics. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 17.

[7] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 142–151. https://doi.org/10.1145/1134285.1134307

[8] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. 2015. Data X-Ray: A Diagnostic Tool for Data Errors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1231–1245. https://doi.org/10.1145/2723372.2750549

[9] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. 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)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[10] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on* 28, 2 (2002), 183–200.