Interactive and Automated Debugging for Big Data Analytics

Muhammad Ali Gulzar
University of California, Los Angeles

ABSTRACT

An abundance of data in many disciplines of science, engineering, national security, health care, and business has led to the emerging field of Big Data Analytics that run in a cloud computing environment. To process massive quantities of data in the cloud, developers leverage Data-Intensive Scalable Computing (DISC) systems such as Google's MapReduce, Hadoop, and Spark .

Currently, developers do not have easy means to debug DISC applications. The use of cloud computing makes application development feel more like batch jobs and the nature of debugging is therefore *post-mortem*. Developers of big data applications write code that implements a data processing pipeline and test it on their local 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 a suspicious result, data scientists spend hours guessing at the source of the error, digging through post-mortem logs. In such cases, the data scientists may want to pinpoint the root cause of errors by investigating a subset of corresponding input records.

The vision of my work is to provide interactive, real-time and automated debugging services for big data processing programs in modern DISC systems with minimum performance impact. My work investigates the following research questions in the context of big data analytics: (1) What are the necessary debugging primitives for interactive big data processing? (2) What scalable fault localization algorithms are needed to help the user to localize and characterize the root causes of errors? (3) How can we improve testing efficiency during iterative development of DISC applications by reasoning the semantics of dataflow operators and user-defined functions used inside dataflow operators in tandem?

To answer these questions, we synthesize and innovate ideas from software engineering, big data systems, and program analysis, and coordinate innovations across the software stack from the user-facing API all the way down to the systems infrastructure.

CCS CONCEPTS

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

KEYWORDS

Debugging and testing, automated debugging, test minimization, fault localization, data provenance, data-intensive scalable computing (DISC), big data, and data cleaning

ACM Reference Format:
Muhammad Ali Gulzar. 2018. Interactive and Automated Debugging for Big Data Analytics. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3183440.3190334

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3190334

1 INTERACTIVE DEBUGGING FOR BIG DATA APPLICATIONS

Developers of DISC applications are notified of runtime failures or incorrect outputs after many hours of wasted computing cycles on the cloud. DISC systems such as Spark do provide execution logs of submitted jobs. However, these logs present only the physical view of Big Data processing and do not provide the logical view of program execution. These logs do not convey which intermediate outputs are produced from which inputs, nor do they indicate what inputs are causing incorrect results or delays, etc. In Spark, crashes cause the correctly computed stages to simply be thrown away which results in valuable computed partial results to be wasted. Finding intermediate data records responsible for a failure corresponds to finding few records in millions, if not billions, of records. The similar problem exists when a user wants to investigate the probable cause of delay in the processing. Finding straggler records are essential for a user to improve the runtime of the application.

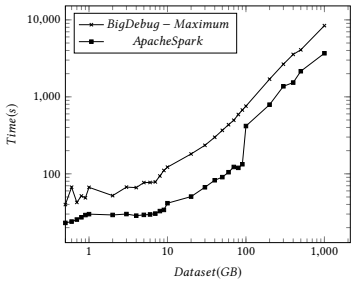

Figure 1: BigDebug's performance and scalability

To address debugging challenges, we design a set of interactive, real-time debugging primitives for big data processing in Apache Spark, the next generation data-intensive scalable cloud computing platform. Our tool BigDebug [5] provides simulated breakpoints, which create the illusion of a breakpoint with the ability to inspect program state in distributed worker nodes and to resume relevant sub-computations, even though the program is still running in the cloud. When a user finds anomalies in intermediate data, currently the only option is to terminate the job and rewrite the program to handle the outliers. To save the cost of re-run, BigDebug allows a user to replace any code in the succeeding RDDs after the breakpoint.

To help a user inspect millions of records passing through a data-parallel pipeline, BigDebug provides on demand guarded watchpoints, which dynamically retrieve only those records that match a user-defined guard predicate which can dynamically be updated on the fly. By leveraging our previous work Titian [6], BigDebug supports fine-grained forward and backward tracing at the level of individual records. To avoid restarting a job from scratch in case of a crash, BigDebug provides a real-time quick fix and resume feature where a user can modify code or data at runtime. It also provides fine-grained latency monitoring to notify which records are taking much longer than other records. BigDebug extends the current Spark UI and provides a live stream of debugging information in an interactive and user-friendly manner.

Benchmark	Dataset (GB)	Overhead	
		Max	w/o Latency
PigMix L1	1, 10, 50, 100, 150, 200	1.38X	1.29X
Grep	20, 30, 40, . . . 90	1.76X	1.07X
Word Count	0.5 to 1000 (increment with a log scale)	2.5X	1.34X

Table 1: Performance Evaluation on Subject Programs

We evaluated BigDebug in terms of performance overhead, scalability, time saving, and crash localizability improvement on three Spark benchmark programs with up to one terabyte of data. With the maximum instrumentation setting where BigDebug is enabled with record-level tracing, crash culprit determination, and latency profiling, and every operation at every step is instrumented with breakpoints and watchpoints, it takes 2.5 times longer than the baseline Spark (see Figure 1). If we disable the most expensive record-level latency profiling, BigDebug introduces an overhead of less than 34% on average as shown in Table 1. BigDebug's quick fix and resume feature allows a user to avoid re-running a program from scratch, resulting in up to 100% time saving. It exhibits less than 24% overhead for record-level tracing, 19% overhead for crash monitoring, and 9% for on demand watchpoint on average.

2 AUTOMATED DEBUGGING OF DISC WORKFLOWS

Errors are hard to diagnose in big data analytics. When a program fails, a user may want to investigate a subset of the original input inducing a crash, a failure, or a wrong outcome. The user (*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 (DP)* (input output record mappings created in individual distributed worker nodes). However, according to our prior study [4], backward tracing based on *data provenance* finds an input set of records 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 [9]. 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.

To find the root cause of the failure from the input dataset in DISC workflows with high precision, we have designed BigSift that brings *delta debugging* (DD) closer to a reality in DISC environments by combining DD with *data provenance* and by also implementing

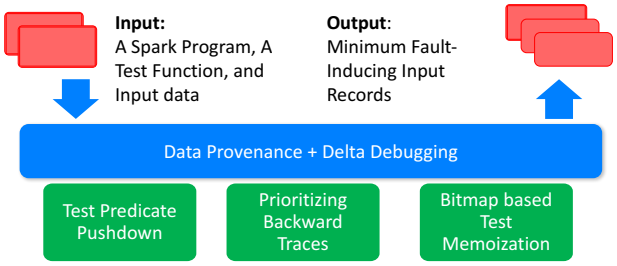

Figure 2: BigSift's Overall Architecture

a unique set of systems optimizations geared towards repetitive DISC debugging workloads (see Figure 2). We re-define data provenance [6] 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. Our current implementation targets Apache Spark [8] but it can be generalized to any data processing system that supports data provenance.

Given a DISC program, an input dataset, and a user-defined test function that distinguishes the faulty outputs from the correct ones, BigSift automatically finds a minimum set of fault-inducing input records responsible for a faulty output in three phases. Phase 1 applies *test driven data provenance* 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* [7]. 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. BigSift augments the current Spark UI and provides a live stream of debugging information.

Running Time (s)		Debugging Time (s)		
Program	Original Job	DD	BigSift	Improvement
Movie Histogram	56.2	232.8	17.3	13.5X
Inverted Index	107.7	584.2	13.4	43.6X
Rating Histogram	40.3	263.4	16.6	15.9X
Sequence Count	356.0	13772.1	208.8	66.0X
Rating Frequency	77.5	437.9	14.9	29.5X
College Students	53.1	235.2	31.8	7.4X
Weather Analysis	238.5	999.1	89.9	11.1X
Transit Analysis	45.5	375.8	20.2	18.6X

Table 2: Fault localization time improvement by BigSift (Programs are explained elsewhere [4])

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 ~10^3 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 [1, 2, 9].

3 WHITE-BOX TEST DATA SAMPLING AND GENERATION IN DISC

Testing big data applications is an expensive and time consuming process because the input data for DISC applications is extremely large. It is clearly infeasible for developers to read through the production data apriori and design test inputs for their application. This problem is exacerbated by the fact that data is originating from diverse sources and is often unstructured, ill-formatted, and schema less. Even though all data records follow the same data-parallel pipeline, many of those records may not share the same program path for each user-defined function. Currently there are no easy means of reducing the size of input data for testing big data applications. Random sampling of data is inadequate, as it may sacrifice test adequacy such as code coverage or fault detection ratios.

During iterative program development, when a developer makes modifications to a dataflow program, he or she may want to select only the subset of data relevant to the modification or determine whether additional records are necessary for testing the modified behavior of the program. During program evolution, when existing input data miss certain program behaviors (i.e., control flow paths), developers must generate new input records to fully test program logic. The interaction of user defined functions and the semantics of data flow operators may make regression testing of data flow programs difficult. Thus, test selection and augmentation techniques must reason about both semantics in tandem.

We propose a new test selection and augmentation technique for DISC applications, BigSample, that improves code statement coverage and fault detection rates of big data analytics applications by reasoning about the semantic of dataflow program directly. BigSample comprises of a new symbolic execution technique for dataflow programs that directly models the path conditions of user-defined functions (UDFs) in conjunction with the semantics of dataflow operators such as join and group-by. By leveraging this symbolic execution engine, BigSample performs test minimization, augmentation, and selection consequently reducing the testing time. To test big data workflows, BigSample first converts the path conditions obtained from UDFs and the specifications of dataflow operators into a test input selection query and produce a subset of input records

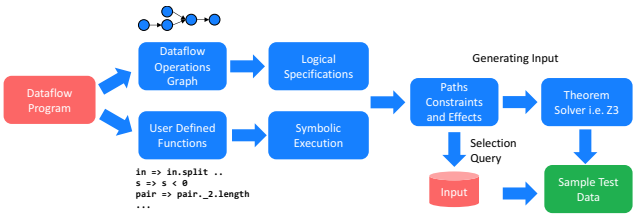

Figure 3: BigSample's Overall Architecture

to be used for iterative development and testing. This test minimization technique can be seen as a new white-box data sampling technique. When the reduced set of records do not cover certain path conditions, BigSample's test augmentation algorithm generates the respective input records by leveraging off-the-shelf theorem provers such as Z3 [3]. This process is also illustrated in Figure 3. Lastly, BigSample performs regression test selection by computing the impact of application logic changes on data.

In terms of evaluation, we plan to assess the efficiency and effectiveness of our test selection and augmentation technique in comparison to two baseline techniques: (1) a testing technique that only reasons about the semantics of data flow operators at a coarse grained level during test selection and augmentation and (2) a naive re-testing technique that re-runs the program on the entire input records. Our investigation will assess testing efficiency in terms of (1) the reduction in the size of original input records that are selected (2) the trade-offs between fault detection rates and the size of selected input records.

REFERENCES

[1] Jong-Deok Choi and Andreas Zeller. 2002. Isolating Failure-inducing Thread Schedules. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, USA, 210–220. https://doi.org/10.1145/566172.566211

[2] Holger Cleve and Andreas Zeller. 2005. Locating Causes of Program Failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. ACM, New York, NY, USA, 342–351. https://doi.org/10.1145/1062455.1062522

[3] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* (2008), 337–340.

[4] 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

[5] 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

[6] 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

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

[8] 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

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