Interactive and Automated Debugging for Big Data Analytics

Muhammad Ali Gulzar
University of California, Los Angeles

ABSTRACT

Data-intensive scalable computing (DISC) systems such as Google's MapReduce, Apache Hadoop, and Apache Spark are being leveraged to process massive quantities of data in the cloud. Modern DISC applications pose new challenges in exhaustive, automatic debugging and testing because of the scale, distributed nature, and new programming paradigms of big data analytics. 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*. DISC applications developers write code that implements a data processing pipeline and test it locally with a small sample data of a TB-scale dataset. They cross fingers and hope that the program works in the expensive production cloud. When a job fails, data scientists spend hours digging through post-mortem logs to find root cause of the problem.

The vision of my work is to provide interactive, real-time, and automated debugging and testing 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 efficacy and efficiency during the development of DISC applications by reasoning about 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.

KEYWORDS

Debugging and testing, automated debugging, test generation, fault localization, data provenance, data-intensive scalable computing (DISC), big data, and symbolic execution

ACM Reference Format:
Muhammad Ali Gulzar. 2019. Interactive and Automated Debugging for Big Data Analytics. In ,. ACM, New York, NY, USA, 6 pages.

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.

Approach. 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 [7] 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.

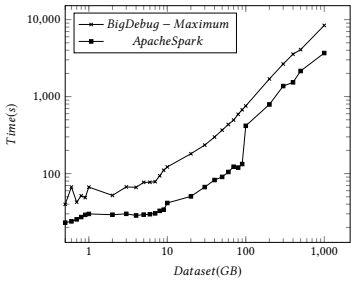

Figure 1: BigDebug's performance and scalability

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 [8], 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

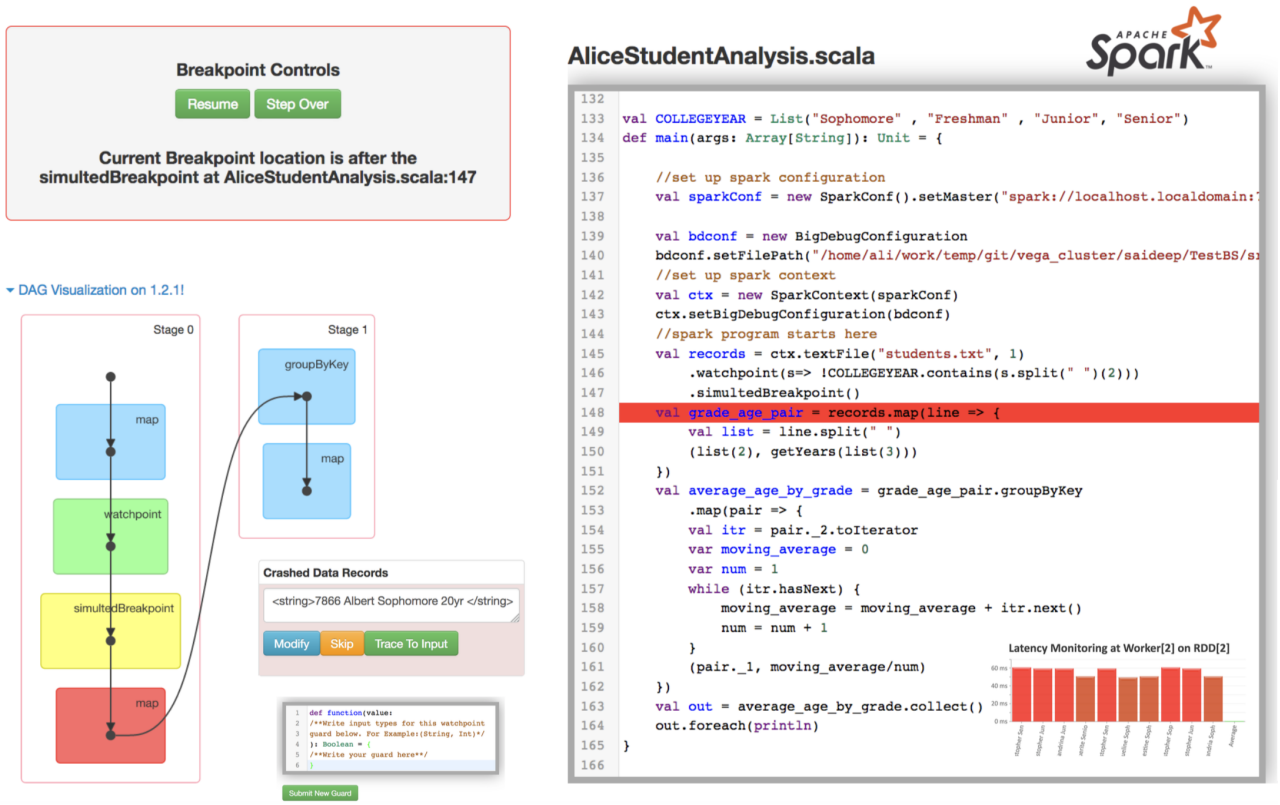

Figure 2: A user can interact with BigDebug's debugging primitives through a web based UI that extends Apache Spark's UI

Spark UI and provides a live stream of debugging information in an interactive and user-friendly manner.

All the features in BigDebug are supported through the corresponding web-based user interface. BigDebug extends the current Spark UI and provides a live stream of debugging information in an interactive and user-friendly manner. A screenshot of this interface is shown in Figure 2.

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

Evaluations. 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 [5], 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 [15]. 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.

Approach. 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 a unique set of systems optimizations geared towards repetitive DISC debugging workloads (see Figure 4). We re-define data provenance [8] for the purpose of debugging by leveraging the semantics of data transformation operators. BigSift then prunes

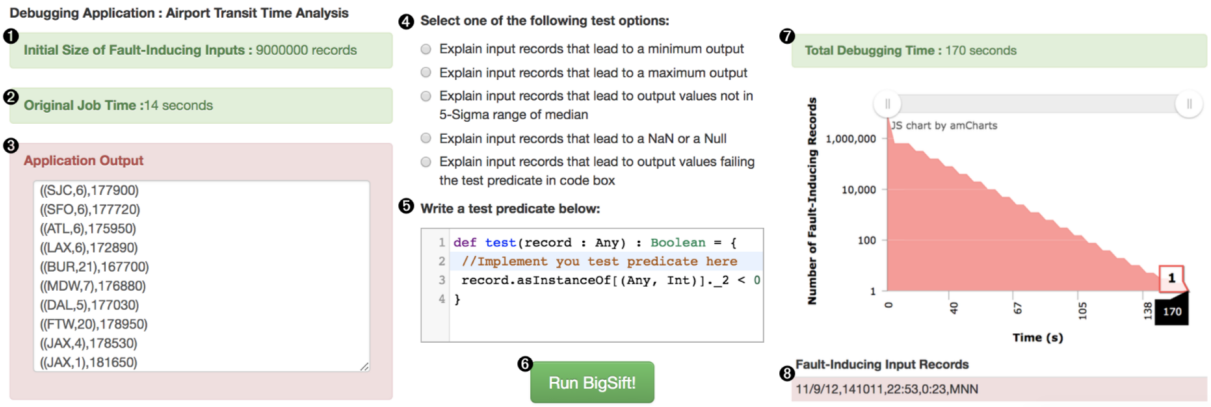

Figure 3: BigSift's User Interface integrated with Apache Spark's application monitoring web service

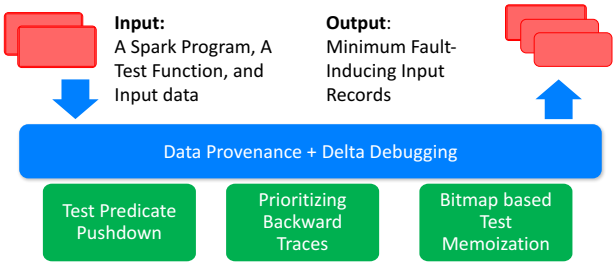

Figure 4: BigSift's Overall Architecture

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.

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* [13]. 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.

Our current implementation targets Apache Spark [14] but it can be generalized to any data processing system that supports data provenance. BigSift is fully integrated with the current Apache Spark's web-based UI as shown in Figure 3. A user can directly inspect raw out-put records (❸), and write a test-oracle function on the fly (❺) or select from pre-defined test oracle functions (❹). BigSift streams real time debugging progress information from the remote cluster to the user through an interactive area plot (❼) and presents the current set of fault-inducing input records in a table format (❽).

Program	Running Time (s) Original Job	Debugging Time (s) 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 [5])

Evaluations. 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 [2, 3, 15].

3 WHITE-BOX TESTING OF BIG DATA ANALYTICS

At *terabyte* scale data processing, rare and buggy corner cases almost always show up in production. Thus, it is common for DISC

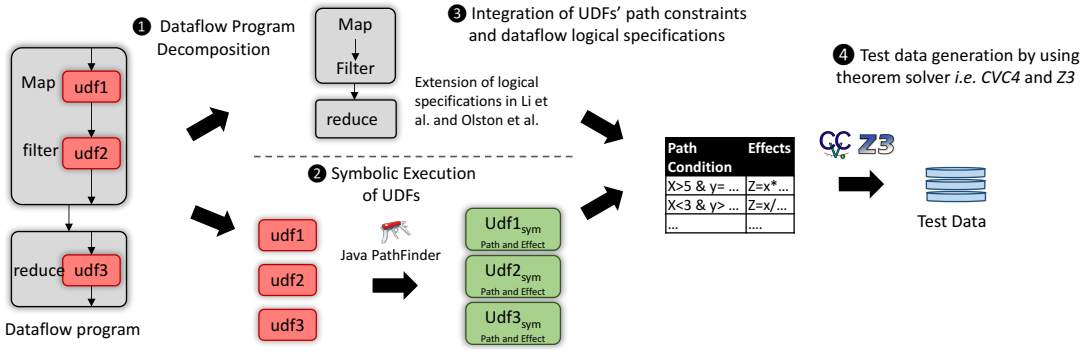

Figure 5: BigTest's Overall Architecture

Dataflow Operators	Olston et al.[10]	Li et al.[9]	BigTest
Load	✓	✓	✓
Map (Select)	✓	✓	✓
Map (Transform)	Incomplete	Incomplete	✓
Filter (Where)	✓	✓	✓
Group	✓	✓	✓
Join	Incomplete	Incomplete	✓
Union	✓	✓	✓
Flatmap (Split)	✗	Incomplete	✓
Intersection	✗	✗	✓
Reduce	✗	✗	✓

Table 3: Support of dataflow operators in related work

applications to either crash after running for days or worse, silently produce corrupted output. Unfortunately, the common industry practice for testing these applications remains running them locally on randomly sampled inputs, which obviously does not reveal bugs hiding in corner cases. We present a systematic input generation tool, called BigTest, that embodies a new white-box testing technique for DISC applications. In order to comprehensively test DISC applications, BigTest reasons about the *combined* behavior of UDFs with relational and dataflow operations. A trivial way is to replace these dataflow operations with their implementations and symbolically execute the resulting program. However, existing tools are unlikely to scale to such large programs, because dataflow implementation consists of almost 700 KLOC in Apache Spark. Instead, BigTest includes a logical abstraction for dataflow and relational operators when symbolically executing UDFs in the DISC application. The set of combined path constraints are transformed into SMT queries and solved by leveraging an off-the-shelf theorem prover, Z3 or CVC4, to produce a set of concrete input records [1, 4]. By using such a combined approach, BigTest is more effective than prior DISC testing techniques [9, 10] that either do not reason about UDFs or treat them as uninterpreted functions.

To realize this approach, BigTest tackles three important challenges. First, BigTest models *terminating* cases in addition to the usual *non-terminating* cases for each dataflow operator. For example, the output of a `join` of two tables only includes rows with keys that match both the input tables. To handle corner cases, BigTest carefully considers terminating cases where a key is only present in the left table, the right table, and neither. Doing so is crucial, as based on the actual semantics of the join operator, the output can contain rows with null entries, which are an important source of bugs. Second, BigTest models collections explicitly, which are created by `flatmap` and used by `reduce`. Prior approaches [9, 10] do not support such operators (as shown in Table 3), and thus are unable to detect bugs if code accesses an arbitrary element in a collection of objects or if the aggregation result is used within the control predicate of the subsequent UDF. Third, BigTest analyzes string constraints because string manipulation is common in DISC applications and frequent errors are ArrayIndexOutOfBoundException and StringIndexOutOfBoundsException during segmentation and parsing.

Approach. BigTest is implemented as a fully automated tool written in Java that takes in a scala-based Apache Spark application as an input and generates test inputs to cover all paths of the program up to a given bound by leveraging an off-the-shelf theorem prover Z3 [4] and CVC4 [1]. Figure 5 illustrates the workflow of BigTest.

In the first step, BigTest compiles a DISC application (comprised of dataflow operators and user-defined functions) into Java bytecode and traverses Abstract Syntax Tree (AST) to search for a method invocation corresponding to each dataflow operator. The input parameters of such method invocation are the UDFs. BigTest stores these UDF as a separate Java class (❶ in Figure 5). For aggregation operators such as `reduce`, the attached UDF must be transformed. For example, the UDF for `reduce` is an associative binary function, which performs incremental aggregation over a collection. We translate it into an iterative version with a loop. To bound the search space of constraints, we bound the number of iterations to a user provided bound `K` (default is 2).

The second step performs symbolic execution on each extracted UDF producing path conditions and effect for each path in the UDF as shown in Figure 5-❷. We make several enhancements in Symbolic Path Finder (SPF)[12] to tailor symbolic execution for DISC applications. DISC applications extensively use string manipulation operations and rely on a `Tuple` data structure to enable key-value based operations. Using an off-the-shelf SPF naïvely on a UDF would not produce meaningful path conditions, thus, overlooking faults during testing. In third step, BigTest combines the path conditions of each UDF with the incoming constraints from its upstream operator leveraging the equivalence classes generated by each dataflow operator's semantics (❸ in Figure 5). BigTest provides logical specifications of all popular dataflow operators with the exception of deprecated operators such as `co-join`.

Finally, BigTest rewrites path constraints into an SMT query. BigTest introduces multiple new symbolic operations to make constraint solving feasible and scalable across the boundaries of string

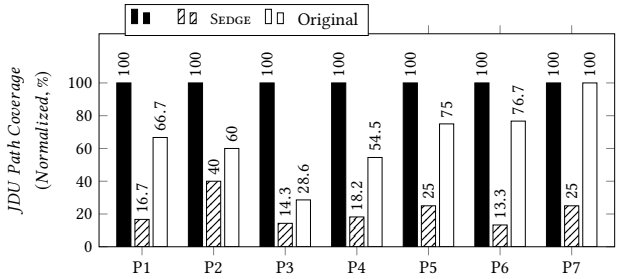

Figure 6: JDU path coverage of BigTest, Sedge, and the original input dataset

	Subject program						
	P1	P2	P3	P4	P5	P6	P7
Seeded Faults	3	6	6	6	4	4	2
Detected by BigTest	3	6	6	6	4	4	2
Detected by Sedge	1	6	4	4	2	3	0

Table 4: Fault detection capabilities of BigTest and Sedge

and number theory. The path conditions produced by BigTest do not contain arrays and instead model individual elements of an array up to a given bound K. BigTest executes each SMT query separately and finds satisfying assignments (i.e., test inputs) to exercise a particular path as shown in Figure 5-❹. Theoretically, the number of path constraints increases exponentially due to branches and loops in a program; however, empirically, our approach scales well to DISC applications, because UDFs tend to be much smaller (in order of hundred lines) than the DISC framework (in order of million lines) and we abstract the framework implementation using logical specifications.

Evaluation. Current benchmarks of DISC applications (PigMix[11], Titian [8], and BigSift [6]), named as P1 to P7 in our evaluation, are a good representative of DISC applications but they do not adequately represent failures that happen in this domain. Therefore, we perform a survey of DISC application bugs reported in Stack Overflow and mailing lists and identify seven categories of bugs. We extend the existing benchmarks by manually introducing these categories of faults into a total of 31 faulty DISC applications. To the best of our knowledge, this is the first set of DISC application benchmarks with representative real-world faults. We assess JDU (Joint Dataflow and UDF) path coverage, symbolic execution performance, and SMT query time. Our evaluation shows that real world DISC applications are often significantly skewed and inadequate in terms of test coverage, when testing them on the entire data set, still leaving 34% of JDU paths untested (see Figure 6). Compared to Sedge [9], BigTest significantly enhances its capability to model DISC applications—In 5 out of 7 applications, Sedge is unable to handle these applications at all, due to limited dataflow operator support and in the rest 2 applications, Sedge covers only 23% of paths modeled by BigTest.

We show that JDU path coverage is directly related to improvement in fault detection—BigTest reveals 2X more manually injected faults than Sedge on average (see Table 4). BigTest can minimize data size for local testing by 10^5 to 10^8 orders of magnitude as shown in Figure 7, achieving the CPU time savings of 194X on average, compared to testing code on the entire production data. Figure 8 shows that BigTest synthesizes concrete input records in 19 seconds on average for all remaining untested paths.

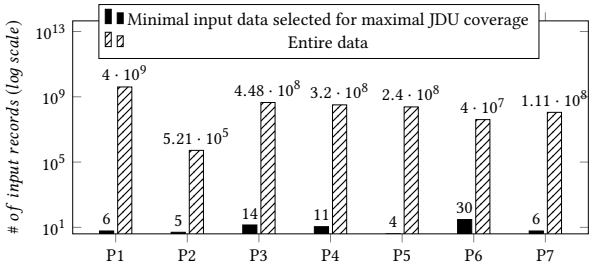

Figure 7: Reduction in the size of the testing data by BigTest

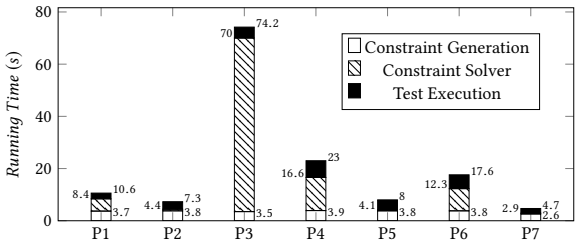

Figure 8: Breakdown of BigTest's running time

Our results demonstrate that *interactive* local testing of big data analytics is feasible, and that developers should not need to test their program on the entire production data. For example, a user may monitor path coverage with respect to the equivalent classes of paths generated from BigTest and skip records if they belong to the already covered path, constructing a minimized sample of the production data for local development and testing.

4 CONCLUSION

Big data analytics are now prevalent in many domains. However, software engineering methods for DISC applications are relatively under-developed. By synthesizing and merging ideas from software engineering and database systems, we design scalable, interactive, and automated debugging and testing algorithms for big data analytics. Our interactive debugging primitives do not compromise the throughput of a cloud application running on a cluster. Through BigSift, we combine data provenance technique from databases and fault-isolation techniques from software engineering to precisely and automatically localize failure-inducing input. To enable efficient and effective testing of big data analytics in real world settings, we present a novel white-box testing technique that systematically explores the *combined* behavior of dataflow operators and corresponding user-defined functions. This technique generates joint dataflow and UDF path constraints and leverages theorem solvers to generate concrete test inputs. We believe that there are more opportunities for adapting existing software engineering methods for big data analytics, e.g. fuzz testing to generate test data for DISC applications, and model checking and runtime verification to help data scientists to have high confidence in the correctness of big data analytics.

REFERENCES

[1] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11) (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz

Qadeer (Eds.), Vol. 6806. Springer, 171–177. http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf Snowbird, Utah.

[2] 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

[3] 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

[4] 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.

[5] 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

[6] 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

[7] 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

[8] 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

[9] Kaituo Li, Christoph Reichenbach, Yannis Smaragdakis, Yanlei Diao, and Christoph Csallner. 2013. SEDGE: Symbolic example data generation for dataflow programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 235–245.

[10] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. 2009. Generating Example Data for Dataflow Programs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. ACM, New York, NY, USA, 245–256. https://doi.org/10.1145/1559845.1559873

[11] K. Ouaknine, M. Carey, and S. Kirkpatrick. 2015. The PigMix Benchmark on Pig, MapReduce, and HPCC Systems. In *2015 IEEE International Congress on Big Data*. 643–648. https://doi.org/10.1109/BigDataCongress.2015.99

[12] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing Nasa Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/1390630.1390635

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

[14] 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

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