Interactive Debugging for Big Data Analytics

Muhammad Ali Gulzar[1], Xueyuan Han[1], Matteo Interlandi[1], Shaghayegh Mardani[1], Sai Deep Tetali[2], Tyson Condie[1], Todd Millstein[1], and Miryung Kim[1]

[1]*University of California, Los Angeles*
[2]*Google, Inc.*

Abstract

An abundance of data in many disciplines has accelerated the adoption of distributed technologies such as Hadoop and Spark, which provide simple programming semantics and an active ecosystem. However, the current cloud computing model lacks the kinds of expressive and interactive debugging features found in traditional desktop computing. We seek to address these challenges with the development of BIGDEBUG, a framework providing interactive debugging primitives and tool-assisted fault localization services for big data analytics. We showcase the *data provenance* and *optimized incremental computation* features to effectively and efficiently support interactive debugging, and investigate new research directions on how to automatically pinpoint and repair the root cause of errors in large-scale distributed data processing.

1 Introduction

To process massive quantities of data in the cloud, developers leverage data-intensive scalable computing (DISC) systems such as Google's MapReduce [7], Apache Hadoop [2], and Apache Spark [30]. In these DISC systems, scaling to large datasets is handled by partitioning data and assigning tasks that execute a portion of the application logic on each partition in parallel. Unfortunately, this critical gain in scalability creates an enormous challenge for data scientists in resolving errors.

The application programming interfaces (API) provided by DISC systems expose a batch model of execution: applications are run in the cloud, and the results, including notification of runtime failures, are sent back to users upon completion. Therefore, debugging is done *post-mortem* and the primary source of debugging information is an execution log. However, the log presents only the *physical view*—the job status at individual nodes, the overall job progress rate, etc., but does not provide the *logical view*—which intermediate outputs are produced from which inputs, what inputs are causing incorrect results or delays, etc. Alternatively, developers may test their program by downloading a small subset of data from the cloud into their local disk, and then run the application in local mode. However, this approach can miss errors when, for instance, the faulty data is not part of the downloaded subset.

To address the above challenges, we develop BIGDEBUG [12], a framework providing expressive and interactive debugging primitives for big data analytics. In BIGDEBUG, we must re-think the notion of breakpoints, watchpoints, and step-through debugging commonly found in a traditional debugger such as `gdb`. For example, simply pausing the entire computation would waste large amounts of computational resources and prevent correct tasks from completing, reducing overall throughput. As another example, requiring the user to inspect millions of intermediate records produced during execution is clearly infeasible. To emulate interactive step-wise debugging without reducing throughput, our *simulated breakpoints* enable a user to inspect a program without actually pausing the entire computation. To minimize unnecessary communication and data transfer, our *on-demand watchpoints* enable a user to retrieve intermediate data using a guard and transfer the selected data on demand. To support systematic and efficient trial-and-error debugging, we enable users to change program logic in response to an error at runtime and *incrementally recompute* affected data only from that step.

In this paper, we showcase the *data provenance* [14] and *optimized incremental computation* [28] features that we have been developing in Apache Spark to support interactive debugging in an effective and efficient manner [12]. We then propose a variety of automated fault localization services that leverage these features together to automatically isolate failure-inducing workflows, diagnose the root cause of an error, and resume the workflow for only affected data and code. BIGDEBUG and the extensions proposed here will contribute to improving

productivity and correctness of big data applications.

2 Background and Motivation

Our effort targets Apache Spark [30], the next generation high-performance distributed dataflow framework, but generalize to other dataflow cloud computing platforms. Apache Spark [3] is a large scale data processing platform that achieves orders-of-magnitude better performance than Hadoop MapReduce [2] for iterative workloads. BIGDEBUG targets Spark because of its support for interactive ad-hoc analytics, allowing programmers to explore the data as they refine their data-processing logic. Furthermore, a variety of domain-specific extensions have been built on Spark [20, 11, 4], which offer unique requirements e.g., for debugging machine learning and graph-based algorithms.

The primary abstraction in Spark is the resilient distributed dataset (RDD) [30], which is a collection of records that can be operated on in parallel. A Spark program is a sequence of operations that input and output RDDs, called *transformations* (e.g., map, reduce, filter group-by, join) and *actions* (e.g., count, collect). Transformations are lazily evaluated; the actual evaluation occurs when an action is called. At that point the Spark runtime executes all the transformations leading up to the action to produce a result. Internally, the Spark master translates a series of RDD transformations into a DAG of *stages*, where each stage contains some sub-series of transformations until a *shuffle step* is required (*i.e.,* data must be re-partitioned).

Suppose that Alice writes a Spark program to parse and analyze election poll logs. The log consists of billions of log entries and is stored in Amazon S3. The size of the data makes it difficult to analyze the logs using a local machine only. Each log entry contains the phone number, the candidate preferred by the callee, the state where the callee lives, and a UNIX timestamp:

```
249-904-9999   Clinton   Texas   1440023983
```

```
1 val log = "s3n://xcr:wJY@ws/logs/poll.log"
2 val text_file = spark.textFile(log)
3 val count = text_file
4       .filter( line =>
            line.contains("Texas"))
5       .filter( line => line.split("
            ")[3].toInt > 1440012701)
6       .map(line = > (line.split(" ")[1] ,
            1))
7       .reduceByKey(_ + _).collect()
```

Figure 1: Election poll log analysis program in Scala

Figure 1 shows the program written by Alice, which totals the number of "votes" in Texas for each candidate, across all phone calls that occurred after a particular date. Line 2 loads the log entry data stored in Amazon S3 and converts it to an RDD object. Line 4 selects lines containing the term 'Texas.' Line 5 selects lines whose timestamps are recent enough. Line 6 extracts the candidate name of each entry and emits a key-value pair of that vote and the number 1. Line 7 counts the votes for each candidate by summing by key.

Alice already tested this program by downloading the first million log entries from the Amazon S3 onto a local disk and running the Spark program in a local mode. When she tests her program with the subset of the data using a local mode, there is no failure. However, when she runs the same program on a much bigger data stored in S3 using a cluster mode, she encounters a crash. Spark reports to Alice the physical view of the crash only—the type of crash, in this case `NumberFormatException`, with a stack trace, the id of a failed task, the id of an executor node encountering the crash, the number of retrials before reporting the crash, etc. However, such physical-layer information does not help Alice to debug which specific input log entry is causing the crash. Even if she identifies a subset of input records assigned to the failed task ID, it is not feasible for her to manually inspect millions of records assigned to the failed task. She tries to rerun the program several times but the crash is persistent, making it less probable to occur due to a hardware failure in the cluster.

3 The BigDebug Framework

BIGDEBUG consists of three main modules: (1) the interactive debugging primitives and API surfaced to users [12]; (2) *Titian data provenance* support for tracing the lineage of records though data-parallel transformations [14]; and (3) *Vega* for *optimized incremental computation* for re-executing modified programs. Titian and Vega are underlying building blocks for making interactive debugging *effective* and *efficient*, since BIGDEBUG must help users to isolate original records relevant to failures only, and trial-and-error debugging often involves re-executing the same program with different inputs or modified program logic.

As a preliminary work, we developed the user-level debug primitives [12] and Titian for data provenance [14]. We are currently developing *optimized incremental computation* support in Spark.

3.1 Interactive Debugging Primitives

Simulated Breakpoint. Doing a step-by-step execution to inspect intermediate outputs is a common debugging strategy. There are several technical challenges in implementing such breakpoints in DISC. First, traditional

breakpoints will pause the entire execution at the breakpoint, while a user investigates an intermediate program state. If we naively implement a normal breakpoint, a driver will communicate with all executor nodes so that each executor will process data until the breakpoint in the DAG and pause its computation until further debug commands are provided. This naive approach causes all the computing resources on the cloud to be temporarily wasted, decreasing throughput. Second, high-performance processing DISC systems such as Spark optimize their performance by pipelining multiple transformations as a single stage. Therefore, there is a mismatch between the logical view of a program and its physical view. Specifically, if two transformations `T1` and `T2` are pipelined as a single stage, then the intermediate results after `T1` are not viewable, as they are not materialized.

To address these challenges, BIGDEBUG provides an illusion of a breakpoint, even though the program is still running on the cloud in the background. For example, when a *simulated breakpoint* is hit, BIGDEBUG may spawn a new process to record the transformation lineage of the breakpoint, while letting the executors continue processing the task. When a user requests intermediate results from the simulated breakpoint, BIGDEBUG recomputes such results on the fly.

On-Demand Watchpoint with Guard. Similar to watching a variable in a traditional debugger `gdb`, a user may want to have a watchpoint to inspect intermediate data. Because millions of records are passing through a data-parallel pipeline, it is infeasible for a user to inspect all intermediate records. Such data transfer would also incur high communication overhead. To overcome these challenges, our *on-demand watchpoint* with a *guard* closure function enables a user to query for a relatively small subset of data matching the guard. A user may also iteratively modify the guard to narrow down the scope of captured data further.

Crash Culprit Determination and Remediation. DISC systems are limited in their ability to handle failures at runtime. For example, crashes in Spark cause the results of all correctly computed stages to simply be thrown away. Remediating a crash at runtime can save time and resources by avoiding a program re-run from scratch. While waiting for a user intervention, it is important to utilize idle resources by running pending tasks continuously to achieve high throughput. To repair the crashed program, a user may want to know not just the current crash culprit record but a crash-inducing input by leveraging Titian's data provenance support in Section 3.2. To avoid the re-computation of stages prior to a crash, BIGDEBUG implements several real-time remediation methods. It enables users to correct the crashed record, skip the crash culprit, or supply a code fix to repair the crash culprit at runtime. This code fix feature can leverage Vega's *incremental computation* (Section 3.3).

BIGDEBUG scales to terabytes and its record-level tracing incurs less than 25% overhead on average. It determines crash culprits orders of magnitude more accurately and provides up to 100% time saving compared to the baseline replay debugger [12].

3.2 Data Provenance for Error Tracing

A common scenario when debugging big data analytics is to first identify the subset of intermediate data that causes a failure, and then deduce which input data is the origin of the failure-causing intermediate records. The second part of this task motivates the need for *data provenance* (also referred to as *data lineage*) support in DISC systems like Spark. While data provenance is well studied in the database community, data provenance support for DISC systems is challenging: operators such as `join` and `group-by` create many-to-one or many-to-many mappings for inputs and outputs, and these mappings are physically distributed across different nodes.

Current approaches to supporting data provenance in DISC systems (specifically RAMP [13] and Newt [18]) cannot support interactive debugging. These systems maintain the provenance metadata in external storage and support data provenance queries through a separate programming interface. In part due to these design choices, they also provide little support for viewing intermediate data or replaying alternative data-processing steps on intermediate data. Our experiments with these approaches (a fragment of which reported in Table 1) show that they do not operate well at scale, which has motivated us to develop our approach.

	Titian		RAMP		Newt	
dataset	grep	wc	grep	wc	grep	wc
500MB	1.27x	1.18x	1.4x	1.34x	1.58x	1.72
5GB	1.18x	1.14x	1.18x	1.32x	2x	3x
50GB	1.14x	1.22x	1.18x	2.20x	7x	28x
500GB	1.10x	1.29x	1.18x	1.60x	14x	inf

Table 1: Overheads of Titian, RAMP and Newt for grep and word count jobs. The overhead is a measure of how much extra time (indicated by a multiplier) is needed to run a job with the data provenance being recorded.

To address this scalability challenge, our work Titian [14] implements data provenance in Apache Spark by directly extending the RDD abstraction with fine-grained data provenance capabilities. From any given RDD, users can obtain a *LineageRDD* reference, which enables *data tracing functionality i.e.,* the ability to transition backward (or forward) in the Spark program dataflow any number of transformation steps. Furthermore, any native RDD transformation can be invoked on a given LineageRDD reference, thereby supporting a re-

play capability. The provenance support provided by our LineageRDD abstraction integrates with Spark's internal batch operators and fault-tolerance mechanisms, offering orders-of-magnitude better performance when compared to prior approaches.

3.3 Optimized Incremental Computation

Program debugging in DISC systems are iterative processes. Programmers start out with an initial workflow, which typically ingests data from several sources, and performs one or more transformations on them. This workflow is then iteratively improved by adding new transformations or modifying existing ones until any observed errors are eliminated and the output has the desired form. Unfortunately, existing DISC systems run each workflow version anew, discarding all work done in the previous iterations. The immense scale of the data typical in cloud computations makes these kinds of development iterations very time consuming, thereby prohibiting the possibility of debugging at interactive scale.

Our on-going work Vega aims to optimize incremental computation in the face of code changes. Vega merges two complementary approaches to speed up the execution of a modified Spark program. First, Vega rewrites a modified program to push the modifications as late as possible in the dataflow, thereby enabling more opportunities to reuse materialized intermediate results from a previous execution. Second, we adapt prior work on *incremental dataflow* [19, 23] to model code changes in terms of data changes and to propagate the deltas to downstream computation. Vega also determines when such delta propagation is more profitable than ordinary re-computation based on a cost estimate.

By mixing the above two approaches, Vega delivers orders of magnitude performance gains with respect to normal Spark in re-executing modified programs. For instance, Figure 3 shows the effectiveness of Vega in evaluating the program of Figure 2, which introduces a map operation that adds a suffix to each word (shown in grey) in a word count program. The re-execution response time under Vega is around three orders-of-magnitude faster than executing from scratch in stock Spark.

```
1 //WordCount.Scala
2 val textFile = spark.textFile("hdfs://...")
3 val counts = textFile
4  .flatMap(line => line.split(" "))
5  .map(word => (word, 1))
6  .map(k, v) => (k + suffix, v)
7  .reduceByKey(_ + _).collect()
```

Figure 2: Edited word count program, the gray box identify the transformation added to the original word count program

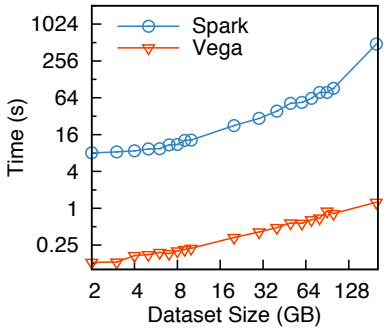

Figure 3: Comparison between Vega and regular Spark for re-evaluating the modified program of Figure 2.

4 Automated Fault Localization Services

Errors are hard to diagnose in big data analytics. An error could occur due to a bug in the program logic, or it could be due to anomalies in the input data. Leveraging the underlying building blocks and mechanisms proposed in Section 3, our vision is to provide tool-assisted capabilities to automatically localize failure-inducing code and data, diagnose the root cause of an error, and safely repair the workflow. This section sketches new research directions on automated fault localization in the context of big data processing.

Defining Test Oracles. The first step is to help users identify failures. Certain types of failures, such as exceptions and occurrences of *NaN* ('not a number') are easy to classify as errors. It is inherently difficult to have expected answers for a vast amount of unseen data. We investigate a variety of methods to allow users to easily define *test oracles*. A user may upload a *ground truth result*, and BIGDEBUG will automatically consider false negatives and false positives as test failures. A user may supply *assertions* and *labeling rules* [15] which can then be used to classify output data as passing or failing.

Isolating Failure-Inducing Inputs. 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. This problem of simplifying and isolating failure-inducing input is a long standing problem in software engineering. *Delta Debugging* (DD) simplifies the set of original inputs to a minimal subset of inputs that still produces the same test failure through systematic testing with different inputs [31]. The DD algorithm splits the original input into different configurations using a binary search strategy and re-runs the same program with different inputs. If all tests with different subset configurations pass, the algorithm increases the granularity of the split by creating even smaller subsets of the selected data. Conversely, if one of the tests fail for a particular subset, it recursively applies the same procedure. The whole process is illustrated in Figure 4.

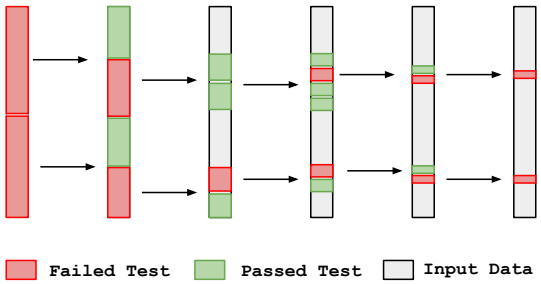

Figure 4: Automated fault localization using Delta Debugging

Applying DD to big data applications would be extremely expensive for two reasons. First, the DD algorithm is a black box procedure and does not consider the structure of the dataflow graph. Therefore, it cannot winnow out irrelevant inputs easily by considering the lineage mapping between intermediate inputs and outputs. Second, the original input is very large and therefore, naively re-running the same program with different inputs subsets can be extremely expensive. To overcome these limitations, we are currently investigating how to combine *Delta Debugging* with *data provenance* and *optimized incremental computation*. Consider a histogram program in Spark that computes an average movie rating for each age group. When the average rating looks suspicious for a particular age group, say the bin with ages from 20 to 30, data provenance can report *all* input records contributing to the particular bin only, winnowing out the records contributing to other bins.

To reduce the cost of reruns, we will investigate how to split intermediate results in half, as opposed to splitting the original input records. For example, by leveraging in-memory default cache at stage boundaries, we can resume the computation from the last transformation right before a wide dependency such as `join` occurs in a dataflow graph. We will also investigate how to further push this data splitting function (essentially a filter) to downstream by analyzing the commutativity of user defined `map` and `filter` functions.

Localizing Faulty Program Logic. We can go even further and localize faults within a transformation of the application-specific program logic. For example, suppose that a user-defined function `word => if (word!=null) (word,1); else (word,0);` is used within a `map` transformation. It may be the case that faults are much more likely when one of the two paths in this function is traversed. To achieve fine-grained, intra-stage fault localization, we will investigate the use of spectra-based fault localization [17, 24, 8]. The challenge of bringing spectra-based fault localization to the space of big data systems is that we must collect path coverage from each distributed worker node in a scalable and efficient manner and aggregate the coverage profile at the driver node to identify faulty program paths.

Repairing Faults Finally, once we have localized the root causes of errors, it is natural to provide tools that aid developers in repairing these errors. The wealth of data available in the setting of big data analytics makes a data-driven approach to repair quite promising. A key challenge is to define a search space of possible program edits that is tractable yet expressive. While theoretically an application can contain arbitrary code, we plan to leverage the fact that a small set of operators appear often (`filter`, `reduce`, etc.) in big data analytics, and each operator has a few common ways in which it is used (e.g., to remove outliers, to perform a summation).

5 Related Work

Fisher et al. [9] interviewed 16 data analysts at Microsoft and studied the pain points of big data analytics tools. Their study finds that a cloud-based computing solution makes it far more difficult to debug and data analysts often dig through trace files distributed across multiple VMs. Zhou et al. [32] manually categorize 210 randomly sampled failures of a big data platform at Microsoft. Job failures and slowdowns are common in DISC applications and many in-field failures are caused by logical and design errors.These findings motivate BIGDEBUG.

Several approaches help developers debug DISC applications by collecting and analyzing execution logs [5, 26, 27, 29, 10]. Unlike BIGDEBUG, none of these help developers debug DISC applications in real time, because they conduct post-mortem log analysis.

Inspector Gadget [21] is a proposal for monitoring and debugging data flow programs in Apache Pig [22]. It simply lists desired debug APIs, but leaves it to others to implement the proposed APIs. Arthur [6] is a post-hoc instrumentation debugger for Spark. It requires a user to write a custom query for post-hoc instrumentation and it supports post-mortem analysis only. Graft [25] is a post-hoc instrumentation debugger for a graph-based DISC computing framework, Apache Giraph [1]. Daphne [16] enables a user to attach a debugger to a remote process on the cluster. Such an approach works for DISC systems that meterialize intermediate results, but it does not work for an in-memory, pipelined, DISC system like Spark.

6 Conclusion

This paper presents the current status and future directions of BIGDEBUG that aims to provide interactive and tool-assisted debugging capabilities for big data analytics. Developers can leverage BIGDEBUG to significantly reduce the amount of debugging time, reducing the overall time to market for their big data applications.

References

[1] Apache giraph. `http://giraph.apache.org/`.

[2] Hadoop. `http://hadoop.apache.org/`.

[3] Spark. `https://spark.apache.org/`.

[4] ARMBRUST, M., XIN, R. S., LIAN, C., HUAI, Y., LIU, D., BRADLEY, J. K., MENG, X., KAFTAN, T., FRANKLIN, M. J., GHODSI, A., AND ZAHARIA, M. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 1383–1394.

[5] BOULON, J., KONWINSKI, A., QI, R., RABKIN, A., YANG, E., AND YANG, M. Chukwa, a large-scale monitoring system. *In Cloud Computing and its Applications (CCA '08)* (10 2008), 1–5.

[6] DAVE, A., ZAHARIA, M., AND STOICA, I. Arthur: Rich post-facto debugging for production analytics applications. Tech. rep., Citeseer, 2013.

[7] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[8] DIGIUSEPPE, N., AND JONES, J. Fault density, fault types, and spectra-based fault localization. *Empirical Software Engineering 20*, 4 (2015), 928–967.

[9] FISHER, D., DELINE, R., CZERWINSKI, M., AND DRUCKER, S. Interactions with big data analytics. *interactions 19*, 3 (May 2012), 50–59.

[10] FU, Q., LOU, J.-G., WANG, Y., AND LI, J. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining* (Washington, DC, USA, 2009), ICDM '09, IEEE Computer Society, pp. 149–158.

[11] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. Graphx: Graph processing in a distributed dataflow framework. *Proceedings of OSDI* (2014), 599–613.

[12] GULZAR, M. A., INTERLANDI, M., YOO, S., TETALI, S. D., CONDIE, T., MILLSTEIN, T., AND KIM, M. Bigdebug: Debugging primitives for interactive big data processing in spark. *To appear in Proceedings of the 38th International Conference on Software Engineering '16*.

[13] IKEDA, R., PARK, H., AND WIDOM, J. Provenance for generalized map and reduce workflows. In *Proc. Conference on Innovative Data Systems Research (CIDR)* (2011).

[14] INTERLANDI, M., SHAH, K., TETALI, S. D., GULZAR, M. A., YOO, S., KIM, M., MILLSTEIN, T., AND CONDIE, T. Titian: Data provenance support in spark. *Proc. VLDB Endow. 9*, 3 (Nov. 2015), 216–227.

[15] INTERLANDI, M., AND TANG, N. Proof positive and negative in data cleaning. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015* (2015), pp. 18–29.

[16] JAGANNATH, V., YIN, Z., AND BUDIU, M. Monitoring and debugging dryadlinq applications with daphne. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on* (2011), IEEE, pp. 1266–1273.

[17] JONES, J. A., HARROLD, M. J., AND STASKO, J. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ICSE '02, ACM, pp. 467–477.

[18] LOGOTHETIS, D., DE, S., AND YOCUM, K. Scalable lineage capture for debugging disc analytics. In *SOCC* (2013), pp. 17:1–17:15.

[19] MCSHERRY, F., MURRAY, D. G., ISAACS, R., AND ISARD, M. Differential dataflow. In *Conference on Innovative Data Systems Research (CIDR)* (2013).

[20] MENG, X., BRADLEY, J., YAVUZ, B., SPARKS, E., VENKATARAMAN, S., LIU, D., FREEMAN, J., TSAI, D., AMDE, M., OWEN, S., XIN, D., XIN, R., FRANKLIN, M. J., ZADEH, R., ZAHARIA, M., AND TALWALKAR, A. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research 2015 abs/1505.06807* (2015).

[21] OLSTON, C., AND REED, B. Inspector gadget: A framework for custom monitoring and debugging of distributed dataflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (2011), ACM, pp. 1221–1224.

[22] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international*

conference on Management of data (2008), ACM, pp. 1099–1110.

[23] PENG, D., AND DABEK, F. Large-scale incremental processing using distributed transactions and notifications. In *OSDI* (2010), vol. 10, pp. 1–15.

[24] REPS, T., BALL, T., DAS, M., AND LARUS, J. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 1997), ESEC '97/FSE-5, Springer-Verlag New York, Inc., pp. 432–449.

[25] SALIHOGLU, S., SHIN, J., KHANNA, V., TRUONG, B. Q., AND WIDOM, J. Graft: A debugging tool for apache giraph. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 1403–1408.

[26] SHANG, W., JIANG, Z. M., HEMMATI, H., ADAMS, B., HASSAN, A. E., AND MARTIN, P. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the 2013 International Conference on Software Engineering* (Piscataway, NJ, USA, 2013), ICSE '13, IEEE Press, pp. 402–411.

[27] TAN, J., PAN, X., KAVULYA, S., GANDHI, R., AND NARASIMHAN, P. Salsa: Analyzing logs as state machines. In *Proceedings of the First USENIX Conference on Analysis of System Logs* (Berkeley, CA, USA, 2008), WASL'08, USENIX Association, pp. 6–6.

[28] TETALI, S. D., INTERLANDI, M., GULZAR, M. A., NOOR, J., CONDIE, T., KIM, M., AND MILLSTEIN, T. Mechanisms for optimizing interactive development of data-intensive applications. *Submitted to a conference*.

[29] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 117–132.

[30] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. 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* (2012), USENIX Association, pp. 2–2.

[31] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on 28*, 2 (2002), 183–200.

[32] ZHOU, H., LOU, J.-G., ZHANG, H., LIN, H., LIN, H., AND QIN, T. An empirical study on quality issues of production big data platform. In *International Conference on Software Engineering, Software Engineering In Practice (ICSE SEIP)* (May 2015), IEEE.