UNIVERSITY OF CALIFORNIA
Los Angeles

Automated Testing and Debugging for Big Data Analytics

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Muhammad Ali Gulzar

2020

ABSTRACT OF THE DISSERTATION

Automated Testing and Debugging for Big Data Analytics

by

Muhammad Ali Gulzar
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2020
Professor Miryung Kim, Chair

The prevalence of big data analytics in almost every large-scale software system has generated a substantial push to build data-intensive scalable computing (DISC) frameworks such as Google MapReduce and Apache Spark that can fully harness the power of existing data centers. However, frameworks once used by domain experts are now being leveraged by data scientists, business analysts, and researchers. This shift in user demographics calls for immediate advancements in the development, debugging, and testing practices of big data applications, which are falling behind compared to the DISC framework design and implementation. In practice, big data applications often fail as users are unable to test all behaviors emerging from interleaving dataflow operators, user-defined functions, and framework's code. *"Testing based on a random sample"* rarely guarantees the reliability and *"trial and error"* and *"printf"* debugging methods are expensive and time-consuming. Thus, the current practice of developing a big data application must be improved and the tools built to enhance the developer's productivity must adapt to the distinct characteristics of data-intensive scalable computing.

By synthesizing ideas from software engineering and database systems, our hypothesis is that we can design effective and scalable testing and debugging algorithms for big data analytics without compromising the performance and efficiency of the underlying DISC framework. To design such techniques, we investigate how we can build interactive and responsive debugging primitives

that significantly reduce the debugging time, yet do not pose much performance overhead on big data applications. Furthermore, we investigate how we can leverage data provenance techniques from databases and fault-isolation algorithms from software engineering to pinpoint the minimal subset of failure-inducing inputs efficiently. To improve the reliability of big data analytics, we investigate how we can abstract the semantics of dataflow operators and use them in tandem with the semantics of user-defined functions to generate a minimum set of synthetic test inputs capable of revealing more defects than the entire input dataset.

To examine the first hypothesis, we introduce interactive, real-time debugging primitives for big data analytics through innovative and scalable debugging features such as simulated breakpoint, dynamic watchpoint, and crash culprit identification. Second, we design a new automated fault localization approach that combines insights from both the software engineering and database literature to bring delta debugging closer to a reality in the big data applications by leveraging data provenance and by constructing systems optimizations for debugging provenance queries. Lastly, we devise a new symbolic-execution based white-box testing algorithm for big data applications that abstracts the implementation of dataflow operators using logical specifications instead of modeling their implementations and combines them with the semantics of any arbitrary user-defined function.

We instantiate the idea of an interactive debugging algorithm as BIGDEBUG, the idea of an automated debugging algorithm as BIGSIFT, and the idea of symbolic execution based testing as BIGTEST. Our investigation shows that the interactive debugging primitives can scale to terabytes—our record-level tracing incurs less than 25% overhead on average and provides up to 100% time saving compared to the baseline replay debugger. Second, we observe that by combining data provenance with delta debugging, we can identify the minimum faulty input in just under 30% of the original job execution time. Lastly, we verify that by abstracting dataflow operators using logical specifications, we can efficiently generate the most concise test data suitable for local testing while revealing twice as many faults as prior approaches. Our investigations collectively demonstrate that developer productivity can be significantly improved through effective

and scalable testing and debugging techniques for big data analytics, without impacting the DISC framework's performance. This dissertation affirms the feasibility of automated debugging and testing techniques for big data analytics—techniques that were previously considered infeasible for large-scale data processing.

The dissertation of Muhammad Ali Gulzar is approved.

Jingsheng Jason Cong

Todd Millstein

Madan Musuvathi

Harry Guoqing Xu

Miryung Kim, Committee Chair

University of California, Los Angeles

2020

To my parents, wife, and siblings

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am very thankful to my Ph.D. advisor, Miryung Kim, for her exceptional support and encouragement throughout the six years of graduate school. I was beyond fortunate to have Miryung, who facilitated my learning and spent countless hours reviewing my drafts and brainstorming new research ideas. Over the years, she paved ways for me to succeed by opening new opportunities through collaborations, research awards, and fellowships. I am profoundly grateful to have worked with her and to be a part of her academic genealogy.

Working with exceptional collaborators has been the most fulfilling experience of my graduate school career. I am honored to have collaborated with Todd Millstein, Tyson Condie, Harry Xu, Madan Musuvathi, Jason Cong, and Quanquan Gu. I am very thankful to them for serving on my Ph.D. committee, providing valuable feedback on my research, and encouraging me with their support. I am grateful to Matteo Interlandi from whom I have learned how to break-apart large-scale systems and for being a good friend who is always available to hear my ideas and give excellent advice. I am also thankful to George Varghese for encouraging me to apply for fellowships and awards, and to Ravi Netravali for helping me with the academic job search. I also want to thank Xiaofeng, Carlos, Nipun, and Biplob for hosting me as a summer intern at Google and NEC Labs.

I would also like to thank my fellow students at UCLA *i.e.,* Tianyi, Saswat, Michael, Seunghyun, Shagha, Jason, Aish, Siva, Qian, John, Fabrice, Lana, Taqi, Lingxiao, Christian, Jiyuan, and Usama for making graduate school a lot more fun and exciting. Furthermore, I would also like to blame many of my friends in the US and Pakistan for making me spend many days traveling with them. It made me feel guilty and, consequently, more productive in research.

Last but not least, I dedicate this dissertation to my parents (Gulzar and Salma), my wife (Ainne), and my siblings (Hina, Mehak, and Hamza). Their unconditional and continuous support has been critical through the ups and downs of graduate school life. Their generosity and understanding have shielded me from all the distresses and distractions of life and helped me focus. If it weren't for them, I would not have been able to achieve this milestone.

The work presented in this dissertation is supported by Google Ph.D. Fellowship, NSF grants CCF-1764077, CCF-1527923, CCF-1460325, CCF-1723773, ONR grant N00014-18-1-2037, Intel CAPA grant, and Samsung grant.

VITA

Sept 2014 – June 2020	Graduate Student Researcher/Teaching Assistant, Department of Computer Science, University of California, Los Angeles.
June 2019 – Sept 2019	Software Engineering Intern, Google Inc, Mountain View CA.
June 2018 – Sept 2018	Software Engineering Intern, Google Inc, Mountain View CA.
June 2016 – Sept 2016	Summer Research Assistant, NEC Labs America, Princeton NJ.
June 2014	B.S. in Computer Science, Lahore University of Management and Sciences, Lahore, Pakistan.

PUBLICATIONS

White-box Testing of Big Data Analytics with Complex User-defined Functions. Muhammad Ali Gulzar, Shaghayegh Mardani, Madanlal Musuvathi, and Miryung Kim. In Proceedings of the 2019 27th ACM Foundations of Software Engineering (ESEC/FSE) 2019.

Automated Debugging in Data-intensive Scalable Computing. Muhammad Ali Gulzar, Matteo Interlandi, Xueyuan Han, Mingda Li, Tyson Condie, and Miryung Kim. In Proceedings of the 2017 Symposium on Cloud Computing (SoCC) 2017.

BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE) 2016.

Perception and Practices of Differential Testing. Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. In Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (SEIP) 2019.

PerfDebug: Performance Debugging of Computation Skew in Dataflow Systems. Jason Teoh, Muhammad Ali Gulzar, Harry Xu, and Miryung Kim. In Proceedings of the 2019 Symposium on Cloud Computing (SoCC) 2019.

LogLens: A Real-Time Log Analysis System. Biplob Debnath, Mohiuddin Solaimani, Muhammad Ali Gulzar, Nipon Arora, Cristian Lumezanu, Jianwu Xu, Bo Zong, Hui Zhang, Guofei Jiang, and Latifur Khan. In 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS) 2018.

Optimizing Interactive Development of Data-Intensive Applications. Matteo Interlandi, Sai Deep Tetali, Muhammad Ali Gulzar, Joseph Noor, Tyson Condie, Miryung Kim, and Todd Millstein. In Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC) 2016.

HeteroRefactor: Refactoring for Heterogeneous Computing with FPGA. Jason Lau*, Aishwarya Sivaraman*, Qian Zhang*, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim. (* are equal co-first authors ordered alphabetically by their last names). In 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE) 2020.

Adding Data Provenance Support to Apache Spark. Matteo Interlandi, Ari Ekmekji, Kshitij Shah, Muhammad Ali Gulzar, Sai Deep Tetali, Miryung Kim, Todd Millstein, and Tyson Condie. The VLDB Journal 2018.

CHAPTER 1

Introduction

Big data analytics has been playing a vital role in the growth of technology and research in many fields, such as health, education, security, and science. For example, in 2013, Twitter analyzed approximately 100 terabytes of data daily to improve its service by recommending the right products, detecting spans, and suggesting follower [95]. Needless to say that this unprecedented growth in data has attracted substantial attention to build systems that are capable of analyzing massive quantities of data. To leverage exiting data center environments, data-intensive scalable computing (DISC) frameworks (*e.g.,* Hadoop MapReduce[52], Apache Spark[145], FlumeJava[35], and Cosmos[34]) run on cloud environments and capable of scaling to petabytes of data. Typically, developers write their big data analytics application on these frameworks in a language like Python, Scala, or Java. The application ingests large-scale data as input, applies a series of operations on the data, and writes the final results in a data storage.

As the popularity of big data analytics grew, the demographics of the big data application developers shifted from expert system builders to business analysts, statisticians, economists, and medical professionals, mostly the people who may not possess to right skills to understand the internals of a DISC system [57]. For such users, the debugging and testing challenges severely impede their progress. Developer productivity tools have been a key area of research for traditional software and have matured over the past three decades. However, recently published experience reports conclude that the *productivity tooling support (especially debugging and testing) has not kept up with the explosive pace of distributed system development* [18, 69, 54].

The following comparison accentuates the distinct characteristics of big data applications and their particular development challenges. Developers typically test their big data application in

a local development workstation with a small sample data, downloaded from a tera-byte scale data warehouse [122]. They cross fingers and hope that the application works in an expensive production cloud. However, big data applications *"rarely work the first time"* [57]. When the application fails, or the results look suspicious, developers inspect giga-bytes of post mortem logs collected across the cloud in order to identify the source of the error. Once the application is fixed, they re-submit the application to the production cloud. This process is repeated until all failing corner cases are adequately handled, and the application terminates successfully with a correct output. Compared to traditional software, debugging big data applications takes several hours while consuming expensive resources in a cloud environment. We list the concrete challenges in this process below.

- *Root-cause analysis is exceedingly expensive due terabyte scale input data*. When a big data application shows symptoms of errors or suspicious output, it is incredibly challenging to isolate the precise root cause among billions of entries in the input.
- *Sample-based testing is inadequate, and full-scale testing is time consuming*. Since the developer has never explored the entire input data, she cannot make any assumptions about it. Using a small sample of the entire input does not provide adequate code coverage, and testing on the entire input data is prohibitively expensive, as each run may take several hours.
- *Big data application executing on the cloud looks vastly different than the one submitted by the user*. To run a big data application on a cloud, DISC frameworks transform the original application by breaking it into computation units called *tasks*. Each of these tasks executes in parallel on each worker node in the cluster to achieve maximum parallelization. Debugging big data applications is time-consuming as the tasks executing on worker nodes look different than the application submitted by the developer.
- *DISC frameworks are complex, with minimum visibility into the application execution*. DISC frameworks comprise of hundreds of complex systems components such as query optimizer, task scheduler, and network communication, each containing millions of code lines. Due to this huge codebase, it is difficult for an average user to comprehend the internal machinery of these frameworks while debugging a big data application.

Traditional debugging and testing techniques in their current state are prohibitively inefficient and ineffective for big data analytics running on the cloud. Conventional breakpoint debugging is infeasible for big data applications, as pausing computations on thousands of worker nodes would be wasteful. Similarly, a variable at a watchpoint may have millions of instances across the cluster that are hard to inspect manually. The post-mortem logs generated by DISC frameworks barely contain insightful debugging information since a sizable portion of these logs reflects physical metrics (*e.g.,* task times, data I/O sizes, memory usage, and GC usage). Even if a user identifies a faulty intermediate data record, it is nearly impossible to trace the fault back to input data manually since the input data might have gone through a series of complex transformation and aggregation operators. Traditional test selection and generation approaches rely on techniques such as symbolic execution to understand the semantics of a program. However, it is infeasible to apply such techniques to the huge codebase of the DISC framework that implements the dataflow operators used in big data applications.

1.1 Problem Formulation

To overcome the limitations of traditional debugging and testing techniques for big data applications, we devise the following hypothesis:

Overall Hypothesis: *By designing interactive and automated debugging and testing techniques that are specifically customized towards big data analytics running on a DISC framework, we can reduce the number of inputs required for testing and reduce time for debugging with minimal performance overhead.*

To design such techniques, we must address three essential user concerns on accelerating the development of reliable big data applications. First, we must provide efficient and interactive debugging primitives that do not compromise the throughput of a big data application running on a large-scale cluster. Second, we must swiftly and automatically isolate the minimum input of a program that resulted in a crash, failure, or suspicious output. Lastly, we must detect most of the

faults efficiently while testing big data applications to avoid setbacks during expensive production runs.

The key insight in this dissertation is that by synthesizing ideas from databases and big data systems research, we can design effective and scalable automated testing and debugging techniques that barely affect the performance and efficiency of the underlying DISC framework. With our hypothesis and key insight, this dissertation evaluates the feasibility of each technique by materializing them into real-world tools that were previously considered infeasible for big data applications. The following sub-sections enumerate this dissertation's contributions individually, drawing up research questions specific to the sub-field alongside the corresponding hypothesis that needs to be validated with thorough evaluations.

1.2 Interactive Debugging Primitives for Big Data Analytics

Debugging of big data application is post-mortem, and the primary source of information is an execution log that does not present the logical view—which intermediate outputs are produced from which inputs. Alternate approaches such as testing on a small subset of big data is ineffective as developers can easily miss errors, for example, when the faulty data is not part of the downloaded subset. To improve current state of debugging of big data applications, we investigate the following hypothesis:

Sub-Hypothesis (SH1): *Based on the deterministic and immutable nature of big data applications, we can design a set of interactive debugging primitives that do not compromise the throughput of the DISC framework running on a cluster and reduce the debugging time significantly.*

We design BIGDEBUG [66] that reinvents the traditional step-through debugging primitives with *simulated breakpoint* which creates the illusion of a breakpoint even though the program is still running in the cloud. It enables users to inspect millions of program states in distributed worker nodes using *guarded watchpoints*, which dynamically retrieve records that match a user-defined

guard predicate. BIGDEBUG also supports fine-grained *forward and backward tracing* at the level of individual record to identify the root cause of a crash. It also provides fine-grained *latency monitoring* to notify a user which records are taking much longer than other records.

We investigate this hypothesis (**SH1**) about interactive debugging by asking three research evaluation questions on performance overhead, scalability, and effectiveness. Over a diverse set of scale-up and scale-out experiments, BIGDEBUG introduces a maximum of 1.5X overhead, a significant proportion of which traces back to record-level latency monitoring. Second, BIGDEBUG has minimum impact on the DISC framework's ability to scale. Lastly, BIGDEBUG reduces the debugging time by 50% compared to the baseline replay debugger. At the production scale, BIGDEBUG has the potential of saving several debugging hours and expensive computing resources.

1.3 Automated Debugging for Big Data Analytics

When a big data application produces a failure or an incorrect result, the developer may want to pinpoint the root cause by investigating the relevant subset of failure-inducing input record among billions of input records. DISC frameworks provide increased expressiveness through user-defined functions, which further increases the complexity of debugging. To address this problem, we pose the following hypothesis:

Sub-Hypothesis (SH2): *A combination of data provenance technique from databases and a test-based fault-isolation technique from software engineering can help developers pinpoint the minimal subset of failure-inducing inputs.*

We design, BIGSIFT [65], that automatically finds a minimum set of fault-inducing input records responsible for a faulty output. It uses test-driven data provenance to prune out input records irrelevant to the given faulty output records, significantly reducing the initial scope of failure-inducing records before applying a test oracle based systematic search called Delta Debugging [147].

Our evaluation shows that BIGSIFT improves the accuracy of faulty inputs by several orders-of-magnitude (~10^3 to 10^7) compared to traditional data provenance approaches from databases and

improves performance by up to 66X compared to Delta Debugging, an automated fault-isolation technique [147]. For each faulty output, BIGSIFT localizes fault-inducing data within 38% of the original job running time, which is previously unheard of, as debugging is expected to take longer than the runtime of a program. By making automated debugging feasible for big data applications, BIGSIFT has the potential to significantly shorten the lifecycle of production bugs.

1.4 White-box Test Generation for Big Data Analytics

At the scale of big data, rare and buggy corner cases frequently show up in production [148]. Unfortunately, the standard industry practice for testing these applications remains running them locally on randomly sampled inputs, which obviously does not flush out bugs hiding in corner cases. Prior studies show that a majority of catastrophic failures in big data applications can easily be prevented by performing local testing on error handling code in the user-defined functions (UDFs) [141]. To address the testing need of big data applications, we investigate the following hypothesis:

Sub-Hypothesis (SH 3): *By abstracting the implementation of dataflow operators and by modeling the semantics of user-defined functions in tandem, we can generate a small set of test inputs that are capable of revealing more defects than the entire input dataset.*

To this end, we design a new white-box test generation approach, BIGTEST [67], that reasons about the *combined* behavior of UDFs with relational and dataflow operations. Instead of symbolically executing the DISC framework's code, BIGTEST models the semantics of dataflow and relational operators using logical specifications and combines them with the symbolic representation of the UDFs. The resulting representation of a big data application is used to produce a miminum set of synthetic input records.

We evaluate BIGTEST on three criteria: fault-detection capability, test data size, and path coverage. Our experiment results show that BIGTEST reveals 2X more manually injected faults than prior approaches for big data applications [91]. The test data generated by BIGTEST is 10^5 to 10^8 orders

of magnitude smaller than the entire dataset, yet it achieves an additional 34% path coverage than the entire dataset. Consequently, BIGTEST achieves the CPU time savings of 194X on average, compared to testing on the entire production data. Through BIGTEST, we show that automated, exhausted, and effective local testing is feasible for big data applications.

1.5 Contributions

We list the contribution of this dissertation below:

- We design the first set of interactive debugging primitives for big data applications in Apache Spark. This work on interactive debugging primitives brings traditional breakpoint and watchpoint based debugging practices to reality without posing significant performance overhead. We realize these concepts in a real-world tool called BIGDEBUG, which is equipped with an interactive user-interface [64].
- We make automated debugging feasible for large-scale big data applications. This work on automated debugging for big data applications isolates the minimum set of fault-inducing inputs among billions of input records. We also built a web-based notebook environment where a developer can inspect the output of a big data application running on the cluster and monitor debugging progress [68].
- We design a symbolic execution-based testing technique for big data analytics. Test data generated by BIGTEST saves several hours of testing effort and expensive cloud resources. We also put forward a benchmark of 35 faults real-world big data applications to support future research in testing and debugging for big data applications [2].

1.6 Roadmap

The rest of this dissertation is organized as follows. Chapter 2 presents related work on debugging and testing practices for traditional software and their counterpart in data-intensive scalable computing. Chapter 3 introduces interactive debugging primitives for large-scale data processing

in Apache Spark. Chapter 4 describes a fully automated approach to isolate the root cause of a failure in a big data application. Chapter 5 describes a new white-box testing technique for big data applications to eliminate the possibility of runtime failures. We conclude this dissertation in Chapter 6 with a vision of future research directions.

CHAPTER 2

Related Work

Data-Intensive Scalable Computing (DISC) systems have become an integral part of a wide variety of production services [95, 127, 124] and are regularly used for day-to-day data analysis tasks performed by data scientists [57, 87, 53]. This widespread adoption is mostly attributed to scalability and performance characteristics of current DISC systems such as Googles MapReduce [52], Apache Spark [145], Microsoft Scope [34]. Due to the widespread usage of DISC systems, the focus of recent research effort shifted to the big data analytics running on top of these systems. Recent progress including end-user declarative languages (*e.g.,* Boom [17], PigLatin [109] DryadLinq [141]) and new visualizations (*e.g.,* Northstar [88] and Wrangler [84]) have lowered the entry barrier for data analysis work [44].

2.1 Challenges in Big Data Analytics

Zhou et al. manually categorize 210 randomly sampled failures of a big data platform at Microsoft [148]. The study finds that 36.2% of in-field failures (*i.e.,* escalations) are caused by system-side defects, including the logical and design errors of big data applications. Job failures and slowdowns are common in big data applications, accounting for 45% and 27% of the escalations. Fisher et al. perform a qualitative study by interviewing 16 data analysts at Microsoft and investigate the pain points of big data analytics tools [57]. Their research finds that a cloud-based computing solution makes it far more difficult to debug. Data analysts often find themselves digging through trace files distributed across multiple VMs, which is often time-consuming and unfruitful. In a very comprehensive survey of user concerns, Bagherzadeh et al. analyse 112K

StackOverflow question posts related to big data [24]. Their investigation discovers that debugging data-centric software is among the top 3 topics of questions (7.4%) outranking performance, which comes at rank 12^{th} with 3.6%.

Kim et al. study 800 data scientists at Microsoft and find the validation of big data analytics to be a major challenge in addition to a lack of techniques that can boost data scientists' confidence in their work [87, 86]. Gunawi et al. analyse 597 unplanned outages of 37 internet services based on cloud computing and find software bugs contributing to 15% of those outages, clearly indicating the inadequate testing [69]. Alvaro et al. publish an extensive experience report on the lack of testing and debugging techniques for distributed systems as today's developers of big data application continue to use GNU GDB-debugger, which should not be the defacto method [18]. Shang et al. elaborate the unique challenges related to verifying and debugging big data analytics executions that can not be addressed easily by traditional approaches [122]. Similar studies on the state of the practices of data-centric software development emphasize the need for testing and debugging methodologies [54, 79, 20]. Gathani et al. interview 20 database users to understand the preferred practices of debugging database queries [59]. Their user interviews reveal that *"printf"* and *"trial and error"* are the standard debugging practices as the users are most familiar with such techniques. All the above mentioned empirical studies on the challenges of developing big data analytics applications motivate this dissertation's focus to design debugging and testing techniques for big data applications.

2.2 Debugging for Big Data Analytics

Execution Log Analysis of DISC Systems. Several approaches help developers debug big data applications by collecting and analyzing execution logs. Boulon et al. monitor Hadoop clusters at runtime and store the log data [115]. Their system collects logs for understanding a runtime failure but does not provide realtime debug primitives. Shang et al. compare the execution log captured on the cloud with the logs obtained using a local mode [122]. Their system abstracts the execution logs, recovers the execution sequences, and compares the sequences between the pseudo-local and

cloud deployments. Tan et al. analyze Hadoop logs to construct state-machine views of the program execution to help a developer understand a Hadoop execution log [125]. Their approach computes the histogram of the duration of each state and detects anomalies in the program execution. Xu et al. parse console logs and combine source code analysis to detect abnormal behavior [139]. Fu et al. map free-form text messages in log files to logging statements in source code [58]. Compared to our work in Chapter 3, none of these post-mortem log analysis approaches help developers debug big data applications in realtime, as their analysis is mostly based on logs comprising of metrics about the system's health (*e.g.,* memory usage, GC, execution times, etc.).

Debuggers for Big Data Applications. Inspector Gadget [108] is a framework proposal for monitoring and debugging data flow programs in Apache Pig [109]. The proposal is based on informal interviews with ten Yahoo employees who write big data applications. While Inspector Gadget proposes features such as step-through debugging, crash culprit determination, tracing, etc., it simply lists desired debug APIs but leaves it to others to implement the proposed APIs. The tracing API of Inspector Gadget targets coarse-grained off-line tracing using a centralized server, falling behind BIGDEBUG's (Chapter 3) ability to trace individual records at runtime.

Arthur is a post-hoc instrumentation debugger that targets Spark and enables a user to replay a part of the original execution [50] selectively. However, a user can only perform *post-mortem* analysis and cannot inspect intermediate results at runtime. It also requires a user to write a custom query for post-hoc instrumentation. To localize faults, Arthur requires more than one run. For example, to remove crash-inducing records from the original input, a program crashes, and Spark reports failed task IDs in the first run. In the second run, a user must write a custom post-hoc instrumentation query (a new data flow graph) with those failed task IDs and run it to recompute the intermediate results for the failed tasks. In the third run, a user removes the crash-inducing records and re-runs the job again. In contrast, BIGDEBUG performs all three steps (*i.e.,* run the applications, debugging the crash-inducing record and re-run the application) in the same run in real-time.

Daphne lets users visualize and debug DryadLINQ programs [83]. It provides a job object

model for viewing the running tasks and enables a user to attach a debugger to a remote process on the cluster. This approach works in DryadLINQ because all communications between tasks are through the disk. Such an approach could work for Hadoop or MapReduce that persist in intermediate results in the file system but does not work for an in-memory processing framework such as Spark. TagSniff introduces a new programming abstraction (*i.e.,* Tag and Sniff) that allows users to insert debugging probes in a big data application [45]. Using Tag API, a user can set a conditional breakpoint, log the intermediate data, or skip the record. The data captured through Tag can further be inspected using a Sniff API. In comparison to our work in Chapter 3, these approaches are not interactive and rely heavily on manual program refactoring by the developer.

Replay Debugger. Replay debugging for distributed systems has been extensively studied [105, 89] through systems such as liblog [60], R2 [70], and DCR [16]. These systems are designed to replay general distributed programs, and thus record all sources of non-determinism, including message passing order across nodes, system calls, and accesses to memory shared across threads. Their goal is to reproduce errors using the captured events. Frameworks like D3S [96], MaceODB [49] and Aguilera et al. [14] are distributed debuggers for finding framework performance bugs, not application bugs. These replay debuggers incur significant overhead at runtime and even larger slowdown at replay time. In contrast, our debugging primitives in Chapter 3 leverage the structure of a data flow graph to replay sub-compensations and re-generate an incremental program snapshot in realtime to support breakpoint debugging, leveraging the deterministic nature of DISC framework.

2.3 Automated Debugging for Big Data Analytics

Data Dependence Analysis For Fault Detection. Detecting bugs in program input by analyzing data dependence has been well explored both in *software engineering* and *databases*. In the database, data provenance (also known as data lineage) is a tool used to explain how query results are related to input data [48]. Herschel et al. recently survey several data provenance techniques of the past and classify them in terms of their use cases, type/granularity of provenance, and sys-

tem logistics [75]. They also propose three challenges in data provenance, one of which is related automation of provenance-based debugging, which this dissertation addresses. Data provenance has been successfully applied both in scientific workflows and databases [74, 47, 27, 19]. However, our experiments in Chapter 4 show that data provenance alone is often unable to compute the minimum input failure-inducing set.

RAMP [112] and Newt [97] added data provenance support to DISC systems; both are capable of performing backward tracing of faults to isolate failure-inducing inputs. However, most data provenance approaches alone are often not able to compute the minimum input failure-inducing set. Chothia et al. [39] is a provenance system implemented over a differential dataflow system, like Naiad [100]. Their approach is more focused on how to provide semantically correct explanations of outputs through replay by leveraging the properties of a differential dataflow system. More recently, Wu et al. design a new database engine, Smoke, that incorporates lineage logic within the dataflow operators and constructs a lineage query as the database query is being developed [113]. With lineage query known upfront, Smoke promises interactive data provenance speed with low overhead. Lamp takes the challenges of data provenance by incorporating the notion of importance through automated differentiation of the program [98]. It is capable of quantifying the importance of inputs through a parallel derivative calculation at runtime. Ikeda et al. present provenance properties such as minimality and precision for individual transformation operators to support data provenance [81, 80]. However, their definition of minimality (minimum provenance) is based on reproducing the same output record rather than producing a faulty output. Therefore, these techniques do not guarantee a minimum set of fault-inducing inputs, whereas our work in Chapter 4, uses the notion of test failure to identify the most precise faulty input.

In software engineering, dynamic taint analysis utilizes information flow to detect security bugs [106, 119] and is also used to perform software testing and debugging [41, 85]. For example, Penumbra leverages dynamic taint analysis to identify failure-relevant inputs automatically [42]. Program slicing is another technique that isolates statements or variables involved in generating a certain faulty output [137, 13, 72]. These techniques use either static and dynamic approaches to

localize relevant code regions. Chan et al. identify failure-inducing input data by leveraging dynamic slicing and origin tracking [36]. In the domain of big data analytics, a single UDF is invoked millions of times on input data records. Recording data lineage information at the granularity of a program statement can easily blow up the storage and add significant runtime overhead. Therefore, traditional taint analysis is not feasible for big data applications.

Automated Debugging Through Repetitive Re-runs. Delta debugging (DD) is a well-known technique for finding the minimal failure-inducing input that causes the program to fail [147, 146, 43]. It has been used for a variety of applications to isolate the cause-effect chain or fault-inducing thread schedules [43, 38]. DD requires multiple executions of the program, which alone is not tractable for DISC system workloads. HDD tries to minimize the number of executions involved in DD by selecting only valid input configuration under the assumption that the input is in a well defined hierarchical structure(*e.g.,* HTML or XML) [103]. In our context, this assumption rarely holds because the input dataset is mostly unstructured. DD is a black-box procedure that does not consider the semantics of data-flow operators and thus cannot prune input records known to be irrelevant. In comparison, our work in Chapter 4 uses data provenance to prune out irrelevant inputs before applying DD, which significantly reduces program re-runs and the debugging time.

Intervention-based Explanation Systems. Several systems have recently addressed the limitations of traditional data provenance to explain anomalous results by computing subsets of the lineage having an "influence" on the outlier result. Systems of this category delete candidate solutions *i.e.,* groups of tuples, from the input and evaluate whether the outlier has changed. This process is called intervention, which is iteratively repeated to find the most influential groups of tuples, usually referred to as explanations [101, 118, 138]. Scorpion finds outliers in the dataset that have the most influence on the final outcome [138]. It restricts itself to queries with aggregates over singles tables (*i.e.,* no joins are involved). It uses predefined partitioning strategies to isolate the most influential partition (configuration of input) and generates the predicates representing this partition of data. Carbin et al. solve the similar problem of finding the influential (critical) regions in the input dataset that have a higher impact on the output using fuzzed input, execution traces,

and classification [33]. Roy et al. mix intervention with causality to overcome the limit of Scorpion in generating explanations over a single table only [118]. Finally, Data X-ray extracts a set of features representing input data properties and summarizes the errors in a structured dataset [136]. It considers the properties of data only and does not reason about how a given program takes the input records and outputs faulty output records. These systems target a specific set of queries and structured data, and therefore their approaches do not apply to generic programs containing, for example, arbitrary UDFs. Furthermore, these approaches are commonly coupled with a DBMS, which limits their scalability.

2.4 Testing For Big Data Analytics

Symbolic Execution. Symbolic execution is a widely used and studied technique in the domain of software engineering [28, 77, 116]. The technique allows a programmer to generate logical constraints that can be used to prove if the program satisfies certain specifications. Tools such as KLEE [30], Pex [128], and JavaPathFinder [135] have brought symbolic execution to the forefront of systematic program testing. In software testing, symbolic execution is traditionally used to discover uncovered program regions in their test cases and use constraint solvers to generate test data that would reveal faults in previously uncovered program region [29, 31, 32, 61, 99, 104, 123].

SQL-integrated applications construct and execute a database query and use the results of a query in the subsequent logic of the program. Popular approaches to test such applications rely on using a variant of symbolic execution to generate both application input and database state. Emmi et al. solve this problem by performing the concolic execution of a program embedded with an SQL query [56]. They symbolically execute the program until the query is executed and replace the query with its pre-defined symbolic constraints. Their approach only caters to a very basic format of an SQL query, which only uses projection, selection, insert and delete operations (*e.g.,* `SELECT ... FROM ... WHERE`). Pan et al. also perform database state generation but with different coverage criteria goals [111]. Their method is similar to the one adopted by Emmi et al. However, they also include support for one more relational operator *i.e.,* `join`. Both

of these techniques cannot deal with big data applications composed of complex and expressive dataflow operators such as `reduceByKey` or `distinct` and arbitrary user-defined functions (see Table 2.1).

Qex is a symbolic execution tool for SQL that follows the traditional symbolic execution based test generation playbook [133]. Qex is loaded with custom theories for each relational operator. For a given SQL query, Qex translates it into an SMT query, which is simplified into constraints using Z3 and used as selection criteria on the input data. Cosette also generates a symbolic representation of a relational query which is used further to either: (a) prove equivalence among two queries or (b) generate test data (counterexamples) that explain conflicting answers from two queries [40]. Both approaches present specifications for dataflow operator that is used by our work in Chapter 5.

One of the most problematic limitations of symbolic execution is the explorations of an enormous number of program paths emerging from the large codebase. In past, several heuristics-based approaches address this path explosion problem [120, 94, 29, 23]. For example, Burnim et al. leverage static analysis to guide symbolic execution towards uncovered paths [29]. Such heuristics could help prioritize specific program paths. As with any other heuristics-based approach, these techniques produce many false negatives as they prioritize exploring certain program paths. Consequently, it may lead to low test quality (or fault detection rate) of the generated test suite. Symbolically executing a big data application in isolation (without the framework codebase) requires abstracting dataflow operator. Our work in Chapter 5 overcomes the limitations of the traditional symbolic execution testing approaches mentioned above by using the logical specification of dataflow operators to generate appropriate symbolic constraints.

Rosette is a framework for designing a solver-aided language [129] to ease the process of translating each language construct into symbolic constraints. This dissertation and Rosette share some similarities in that they both translate higher-order types such as tuples or arrays into lower-level constraints. For side-channel analysis, Bang et al. address a similar problem (as of this dissertation) of solving constraints that cross boundaries between different theories (numerics, integer, and string constraints) [25]. Such cross-theory constraints are known to be challenging

to solve with Z3 [51] or CVC4 [26]. They extend SPF by modeling strings into bit-vectors and by integrating numeric model counting in ABC [22], an automata-based string constraint solver, to perform model counting for both numeric and string constraints. Such extensions of ABC and SPF could be used to generate satisfying assignments for string-based constraints in the constraints solving phase of BIGTEST, presented in Chapter 5.

Regression Test Selection and Augmentation. Regression testing has been extensively studied in software testing. Safe regression testing allows users to select only those test cases that exercise the updated regions of a program and reveal all possible faults associated with these updates. Rothermel et al. summarize several regression testing techniques and evaluate them under a controlled environment [117]. Their evaluation is based on four key metrics: (1) *inclusiveness*, measuring the extent by which the chosen test cases lead to a different output; (2) *precision*, measuring the ability of a technique to avoid selecting test cases that would not lead to a different output; (3) *efficiency*, quantifying the practicality of the technique in terms of its computational cost; and (4) *generality* checking if a technique is applicable on a diverse set of language constructs. Harrold et al. use the difference between the control flow graphs (CFGs) of two versions of a program to select the test cases most relevant to program modifications [73]. For each test case, this technique requires a list of statements or CGF nodes covered by the test case.

Test prioritization helps the developers run their test cases to maximize fault detection in a minimum amount of time. Elbaum et al. evaluate 14 different ways of prioritizing test cases to achieve a higher rate of fault detection [55]. Their work prioritizes a test case on the metrics of code coverage, fault exposing potential, or fault proneness. Several test prioritization methods may be reused to prioritize input data records in big data applications under testing to reach the maximum coverage. However, almost all of the above techniques require coverage profile or fault, exposing the potential for each input record. As mentioned earlier, collecting and generating such information is infeasible and expensive at large scale. However, using the logical specification of dataflow operators defined in Chapter 5, we can abstract the coverage profile of the framework's code in terms of the equivalence classes of corresponding operators.

Testing Large Scale Systems. Alvaro et al. publish an experience report on the state of testing of distributed systems [18], finding existing approaches to be ad-hoc, suitable for highly skilled ”superuser”, and fundamentally unscalable. They put forward a motion for the systems community to build end-user testing tools to increase the faults tolerance of distributed software (*e.g.,* big data applications), resonating with the goal of this dissertation. Our prior empirical study on the use of differential testing for large-scale, end-to-end systems instead of traditional unit testing and find unit testing either incomplete or infeasible in practice due to the system's complexity [63]. They further observe that more than 40% of tests in real-world production software take between 15 minutes to several hours, stressing fuzz testing infeasibility on large-scale, long-latency systems. Other studies at Microsoft and Google concur exceedingly long-running test times (in the order of hours) on their products, such as Microsoft Windows [76, 132].

Testing SQL and Data Analytics. Miao et al. generate database rows to explain the output difference between the two queries [102]. They leverage hard-coded specifications of relational operators and model their impact on the data provenance via input constraints. Such constraints are used to generate synthetic data, facilitating the reasoning behind output difference among two queries. Gupta et al. pivot their test generation technique for SQL queries guided by mutation testing of relational operators [71]. They define a set of mutations for selected relational operators such as join and specify rules needed for each type of join to kill the mutant. Relational database applications rarely use user-defined function (UDF) that are prevalent in big data applications. Even though the tools mentioned above assist in identifying faults in the relational query, they may not reveal the faults emerging from UDF alone and from the interaction of UDF and dataflow operators.

Testing Map-Reduce Programs. Csallner et al. propose the idea of testing commutative and associative properties of Map-Reduce programs by generating symbolic constraints [46]. Their goal is to identify non-determinism in a Map-Reduce program that may arise because of a non-associative or a non-commutative user-defined function in a `reduce` operation. They also produce counterexamples as evidence by running a constraint solver over symbolic path constraints. Xu et

Dataflow Operators	Olston et al. [107]	Li et al. [91]	Emmi et al. [56]	Pan et al. [111]
Load	✓	✓	✓	✓
Map (Select)	✓	✓	✓	✓
Map (Transform)	✓	✓	✗	✗
Map (Non-Invertible)	✗	✗	✗	✗
Filter (Where)	✓	✓	✓	✓
Group	✓	✓	✗	✗
Join	✓	✓	✗	✓
Union	✓	✓	✗	✗
Distinct	✗	✓	✗	✗
Flatmap (Split)	✗	✓	✗	✗
Intersection	✗	✗	✗	✗
Sort	✗	✗	✗	✗
Reduce	✗	✗	✗	✗
Cartesian	✗	✗	✗	✗
Coalesce	✗	✗	✗	✗

Table 2.1: Support of dataflow operators in related work

al. extend the work by Csallner et al. by adding a few more Map-Reduce program properties [140]. In addition to non-determinism, they also test (1) *selectivity*, to verify the ratio between the size of an input and an output of an operator; (2) *statefulness*, to verify if the current computation of an operator depends on a previous computation; and (3) *partition interference* to know if other partitions interfere with the results in the current partition. Unlike BIGTEST (Chapter 5), both of these techniques only test high-level properties of individual dataflow operators used in a Map-Reduce program. These approaches do not provide adequate coverage of user-defined functions and overlook logical faults in a big data application.

Li et al. propose a combinatorial test data generation approach that extracts input domain information automatically from data schema or original input and bound the scope of possible input combinations [92]. Such an approach can easily blow up with an excessive number of input combinations for input fields with relaxed constraints such as price or first name). A more generalized work by Olsten et al. test big data applications written in Pig Latin by generating testing data [107]. Compared to random sampling, their approach provides concise data selected from an input dataset that promises the full statement coverage of a big data application. They model each dataflow operators into a set of equivalence classes. For example, a filter operator translates into two equivalence classes where the first-class contains data records passing the filter, and the second one includes the records failing the filter. The dataflow operators they support are listed in Table 2.1. Their approach does not consider the semantics of UDFs directly and fails to generate

test data when applied to non-invertible UDFs in operators such as `transform`.

Li et al. extend Olsten et al's approach in their tool SEDGE by supporting additional dataflow operators such as `distinct` and `split`. They use symbolic execution on dataflow operators to produce path constraints. SEDGE considers UDFs as black-box procedures and uses them to generate data for uncovered equivalence classes. The high-level constraints generated by their tool may not provide the full statement coverage of the UDFs, which are crucial in testing expressive big data applications such as Spark programs. Table 2.1 shows a side by side comparison of the related work in terms of dataflow operators which they support.

In practice, local testing of big data application is popular and, to a certain extent, shows promise in detecting code faults. However, it is critically important to have the right test input data to expose program behaviors that usually surface only at scale. Yuan et al. study 198 user-reported failures on popular DISC systems and find that majority of the failures in these reports can be reproduced with three or fewer nodes, showing the efficacy of local testing [142]. Li et al. perform an extensive analysis of failure in big data applications at Microsoft [93]. A key conclusion of their study is to improve the state of local testing using a small yet carefully selected/generated subset of input. In Chapter 5, we follow up on this recommendation and introduce a test generation technique for big data application, making local testing highly effective.

CHAPTER 3

Interactive Debugging Primitives for Big Data Analytics

Debugging the massive parallel computations that run in today's data-centers is time-consuming and error-prone. This chapter explores the following research question on debugging: *what are the necessary debugging primitives for interactive big data analytics?* To address this question, we investigate the sub-hypothesis **SH1**: *Based on the deterministic and immutable nature of big data applications, we can design a set of interactive debugging primitives that do not compromise the through-put of the DISC framework running on a cluster and reduce the debugging time significantly.* In this chapter, we re-think the traditional step-through debugging and propose a set of interactive, expressive, and real-time debugging primitives for big data applications, meeting the requirements of low overhead, scalability, and fine granularity.

3.1 Introduction

Currently, developers do not have easy means to debug big data applications. The use of cloud computing makes application development feel more like batch jobs, and the nature of debugging is, therefore, *post-mortem*. Developers are notified of runtime failures or incorrect outputs after many hours of wasted computing cycles on the cloud.

We design BIGDEBUG to provide interactive, real-time debugging primitives for big data applications. BIGDEBUG provides simulated breakpoints and watchpoint, which create the illusion of a breakpoint with the ability to inspect selected program state in distributed worker nodes. BIGDEBUG also supports fine-grained forward and backward tracing at the level of individual records by leveraging prior work on data provenance within Spark [82]. To avoid restarting a job from scratch

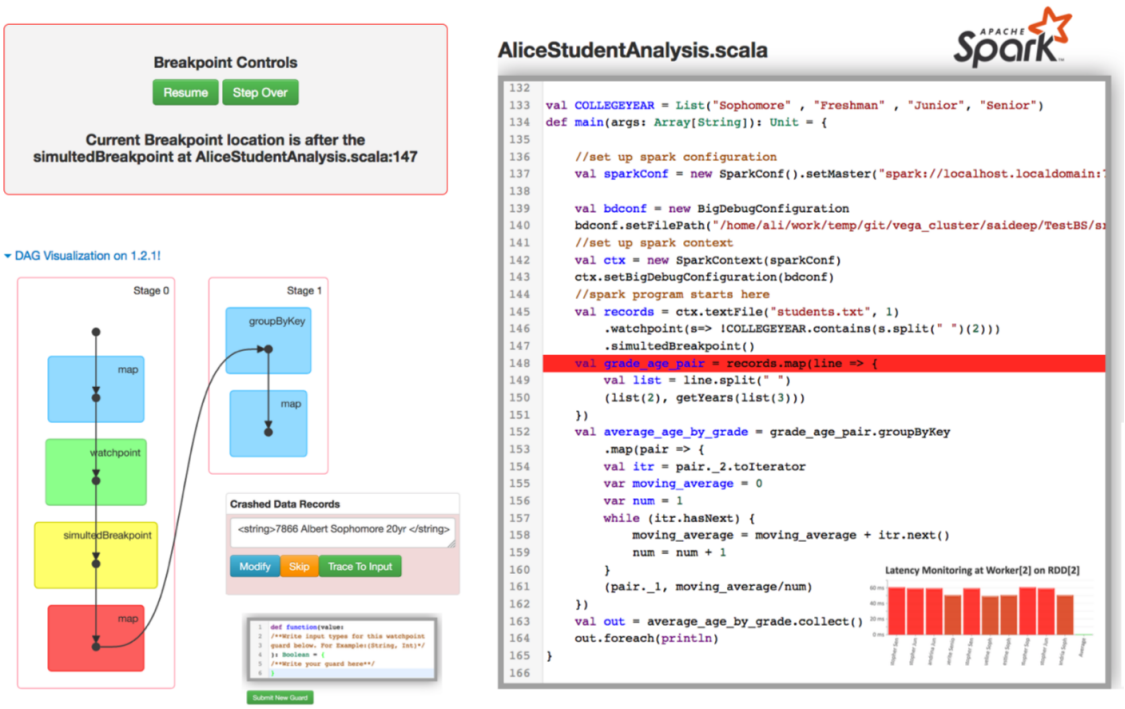


Figure 3.1: BIGDEBUG's web-based interactive user interface

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 a user which records are taking much longer than other records. All the features in BIGDEBUG are available through a web-based user interface, as shown in Figure 3.1, which provides a live stream of debugging information.

In the most conservative experimental setting, BIGDEBUG takes 2.5 times longer than the baseline Spark 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. If we disable the most expensive record-level latency profiling, BIGDEBUG introduces an overhead less than 34% on average. BIGDEBUG's quick fix and resume feature allows a user to recover appropriately from a crash and resume the rest of the computation, resulting in up to 100% time-saving. BIGDEBUG narrows down the scope of failure-inducing data by orders of magnitude through fine-grained tracing of individual records within the distributed data processing pipeline.

The rest of this chapter is organized as follows: Section 3.2 discusses the background on large

data-parallel processing in Apache Spark and why we chose Spark. Section 3.3 introduces a set of interactive debugging primitives with a running example. Section 3.4 describes the design and implementation of individual features. Section 3.5 presents a demonstration of BIGDEBUG. Section 3.6 describes evaluation settings and the results. Section 3.7 discusses the next research direction.

3.2 Background: Apache Spark

Apache Spark [145] is a large scale data processing platform that achieves orders-of-magnitude better performance than Hadoop MapReduce [3] for iterative workloads. BIGDEBUG targets Spark because of its wide adoption—with over 800 developers and 200 companies leveraging its capabilities—and support for interactive ad-hoc analytics, allowing programmers to explore the data as they refine their data-processing logic. Spark's high-level programming model provides over 80 types of data manipulating operations and supports language bindings for Scala, Java, Python and R. Furthermore, a variety of domain specific extensions have been built on Spark, including MLLib [4] for machine learning, GraphX [62] for graph processing, SparkSQL [21] for relational queries, and statistical analysis in R. Spark can consume data from a variety of data sources, including distributed file systems (like HDFS [121]), object stores (like Amazon S3 [1]), key-value stores (like Cassandra and HBase), and traditional RDBMS (like MySQL and Postgres).

The Spark programming model can be viewed as an extension to the MapReduce programming model that includes direct support for traditional relational algebra operators (e.g., group-by, join, filter), and iterative computations through a "for" loop language construct. These extensions offer orders-of-magnitude better performance over previous Big Data processing frameworks like Apache Hadoop [3] for iterative workloads like machine learning. Spark also comes with a relaxed fault tolerance model—based on workflow lineage [27]—that is built into its primary abstraction: Resilient Distributed Datasets (RDDs) [144], which exposes a set of data processing operations called **transformations** (e.g., map, reduce, filter group-by, join) and **actions** (e.g., count, collect).

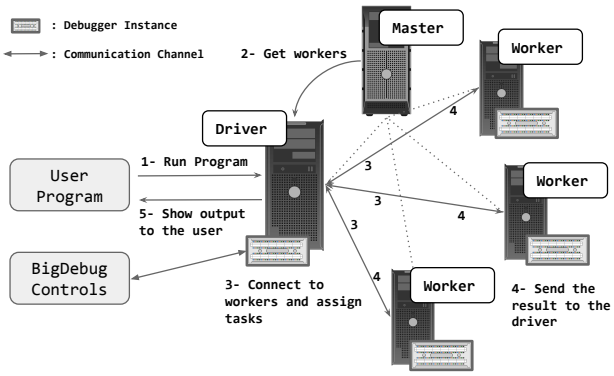


Figure 3.2: Architecture of Spark with BIGDEBUG

Spark programmers leverage RDDs to apply a series of transformations to a collection of data records (or tuples) stored in a distributed fashion *e.g.,* in HDFS [121]. Calling a transformation on an RDD produces a *new* RDD that represents the result of applying the given transformation to the input RDD. Transformations are lazily evaluated. The actual evaluation of an RDD occurs when an action is called. At that point the Spark runtime executes all transformations leading up to the RDD, on which it then evaluates the action e.g., the `count` action counts the number of records in the RDD. A complete list of transformations and actions can be found in the Spark documentation [6].

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

Figure 3.3: Scala word count application in Apache Spark

Figure 3.3 shows a word count program written in Spark using Scala. In this program, the frequency of each unique word in the input text file is calculated. It splits each word using a space as a separator, and maps each word to a tuple containing the word text and 1 (the initial count). The `reduceByKey` transformation groups the tuples based on the word (i.e., the key) and sums up the word counts in the group. Finally, the `collect` action triggers the evaluation of the RDD referencing the output of the `reduceByKey` transformation. The result of the `collect` action is a list of tuples—containing each unique word and its frequency—that is returned to the driver program.

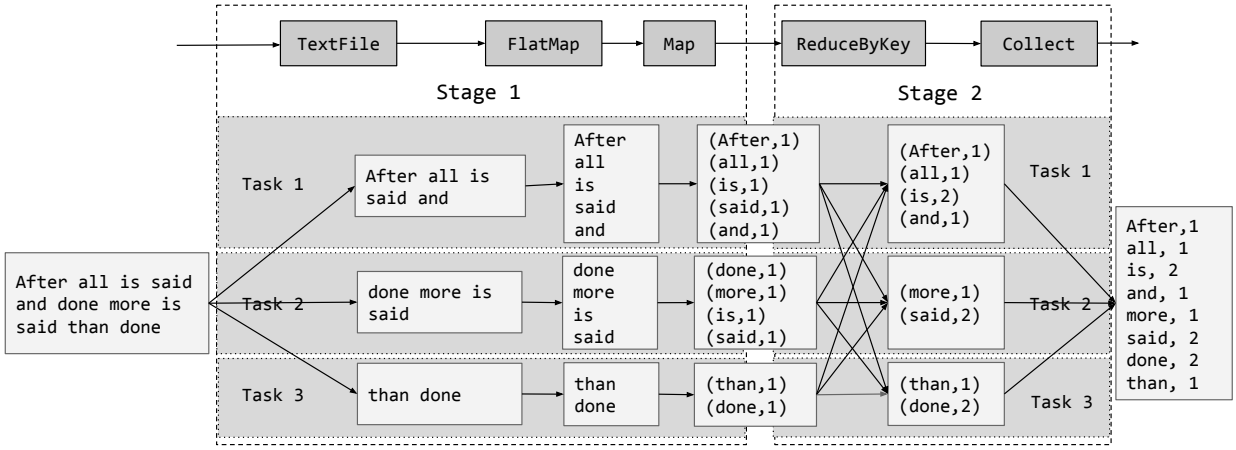


Figure 3.4: Data transformations in word count with 3 tasks

The Spark platform consists of three modules: a driver, a master, and a worker. A master node controls distributed job execution and provides a rendezvous point between a driver and the workers. The master node monitors the liveliness of all worker nodes and tracks the available resources (i.e., CPU, RAM, SSD, etc.). Worker nodes are initiated as a process running in a JVM. Figure 3.2 shows an example Spark cluster containing three worker nodes, a master node, and a driver. A Spark job consists of a series of transformations that end with an action. Clients submit such jobs to the driver, which forwards it to the master node. 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). The Spark scheduler is responsible for executing each stage in topological order, with *tasks* that perform the work of a stage on input partitions. Each stage is fully executed before downstream dependent stages are scheduled. The final output stage evaluates the action that triggered the execution. The action result values are collected from each task and returned (via the master) to the driver program, which can initiate another series of transformations ending with an action.

Figure 3.4 represents the execution plan (stage DAG) for our word count example in Figure 3.3. The input text is divided into three partitions. The driver compiles the program into two stages. Stage 1 applies the `flatmap` and `map` transformations to each input partition. A shuffle step is then required to group the tuples by the word. Stage 2 processes the output of that shuffle step by

```
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 3.5: Election poll log analysis program in Scala

summing up the counts for each word. The final result is then collected and returned to the driver. In this example, both stages are executed by three tasks; in Stage 1, a task is assigned to each input partition; and in Stage 2, a task is assigned to each partition produced by the shuffle step. It is also worth noting that each task runs on a separate thread, so each worker may run multiple tasks concurrently using multiple executors based on resource availability such as the number of cores.

3.3 Motivating Scenario

This section overviews BIGDEBUG's features using a motivating example. 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, for example:

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

Figure 3.5 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

```
1  filter{ line =>
2    var time = line.split(" ")[3]
3    val date = new Date("YYYY-MM-DD")
4    if( date.checkFormat( time ) )
5      time = date.getTimeInUnix( time )
6    time.toInt > 1440012701 }
```

Figure 3.6: Realtime code fix for filter

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 re-trials 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. Though Spark reports the task ID of a crash, it is impossible for Alice to know which records were assigned to the crashed executor and which specific entry is causing the crash. Even if she identifies a subset of input records assigned to the task, 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.

Crash Culprit Determination. With BIGDEBUG, Alice is provided with a specific record causing the crash, in this case a record, "312-222-904 Trump Illinois 2015-10-11." BIGDEBUG first reports a specific transformation responsible for the crash as well—line 5 in Figure 3.5 at `time.toInt`, which tries to change the timestamp from String to Integer, causing a `NumberFormatException`. Using BIGDEBUG's backward tracing feature, Alice then locates the specific log entry in the S3 input causing the crash. She then sees that this log entry uses a timestamp in Date format rather than UNIX format.

RealTime Code Fix and Resume. Without BIGDEBUG, Alice can only modify the input data and restart the job from scratch, incurring wasted computation in running Amazon EC2 services. Using BIGDEBUG, Alice can fix the code on the fly by replacing the original filter at line number 5 with the one in Figure 3.6 and resuming the failed computation.

Guarded Watchpoint. Even after fixing the crash, Alice sees that the total number of votes found by the program is greater than what she expected to find. Using BIGDEBUG's breakpoint, Al-

```
1  abstract class RDD[T: ClassTag]( ....
2  def watchpoint(f: T => Boolean): RDD[T]
3  def breakpoint
4  def breakpoint(f:T => Boolean)
5  def enableLatencyAlert(set : Boolean)
6  def setCrashConfiguration(set : CrashConfiguration)
7  def setFunction(f : T => U)
8  def goBackAll: LineageRDD
9  def goNextAll: LineageRDD
10 def goBack: LineageRDD
11 def goNext: LineageRDD
12 ....
```

Figure 3.7: BIGDEBUG's API

ice investigates the intermediate result right after the second transformation at line number 5 in Figure 3.5. Without BIGDEBUG, investigating such intermediate result is not possible, because Spark combines all transformations within a single stage and evaluates them all at once. BIGDEBUG allows a user to set a breakpoint at any transformation step and investigate the intercepted intermediate results. She suspects that some UNIX timestamps may be in the 13-digit millisecond format, while the code assumes timestamps are in the 10-digit second format. She therefore installs a watchpoint guarded by the following predicate (an ordinary function): `(line => line.split(" ")[3].length > 10`. BIGDEBUG dynamically retrieves the data matching this guard, and she can continue to modify the guard iteratively in order to investigate further.

3.4 Debugging Primitives

To provide interactive step-wise debugging primitives in Spark, BIGDEBUG must address three technical challenges. First, it must be **scalable** to handle large data sets on the order of terabytes. Second, since the debugger process on the driver must monitor and communicate with a large number of worker nodes performing tasks on the cloud, BIGDEBUG must have a **low overhead**, minimizing unnecessary communication and data transfer. Third, to help localize the cause of errors, BIGDEBUG must support **fine-grained data inspection** and monitoring capabilities at the level of individual records rather than tasks. Currently, Spark reports failures at the level of tasks only. Since a single task processes millions of records, locating a failed task is inadequate, as it is impossible for a user to manually inspect millions of records.

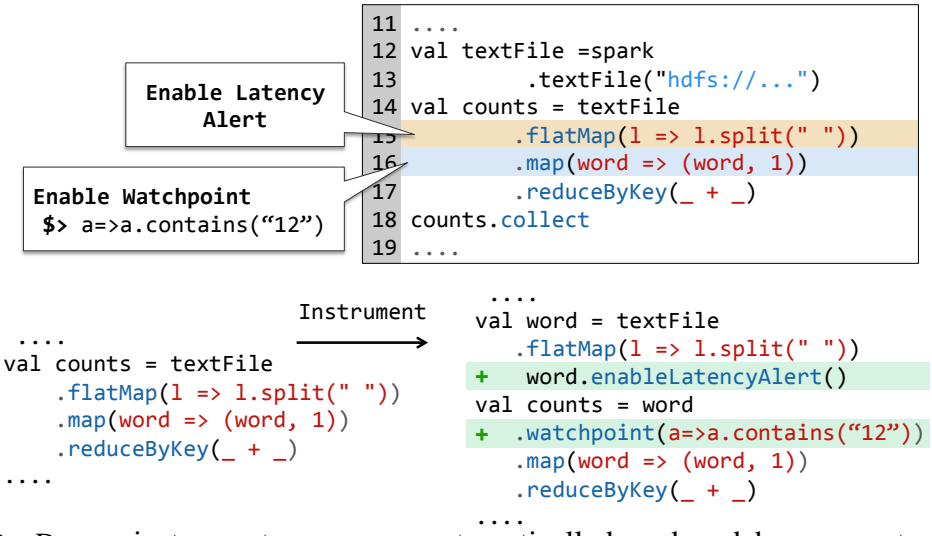


Figure 3.8: BIGDEBUG instruments a program automatically based on debugger control commands entered by a user.

BIGDEBUG tackles these challenges by adopting a tight integration approach with Spark's runtime. Instead of creating a wrapper of existing Spark modules to track the input and output of each stage, BIGDEBUG directly extends Spark to monitor pipelined, intra-stage transformations. To maximize the throughput of big data processing, BIGDEBUG provides *simulated breakpoints* that enable a user to inspect a program state in a remote executor node without actually pausing the entire computation. To reduce developers' burden in inspecting a large amount of data, *on-demand watchpoints* retrieve intermediate data using a guard and transfer the selected data on demand. These primitives are motivated by prior user studies in Inspector Gadget [108], where they interviewed DISC developers in Yahoo and found that DISC developers want step-through debugging, crash culprit determination, and tracing features. Inspector Gadget proposes desired primitives but does not implement them.

The API for BIGDEBUG is shown in Figure 3.7 and targets Scala. A user may also use a web-based debugger UI to automatically insert corresponding API calls in the code. For example, the instrumented code on the bottom-right of Figure 3.8 is automatically generated from the debugger commands at the top of the figure.

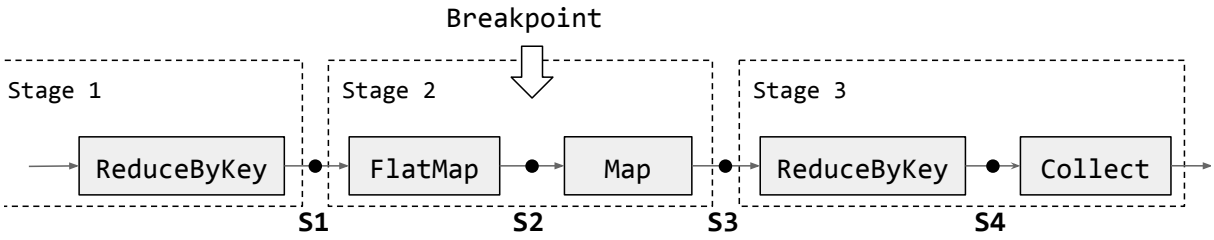


Figure 3.9: Illustration of simulated breakpoint

3.4.1 Simulated Breakpoint

Doing a step by step execution to inspect intermediate outputs is a common debugging strategy. There are two technical challenges in implementing breakpoints in Spark. 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, Spark optimizes its performance through pipelining transformations in a single stage. Therefore there is a mismatch between the logical view of data transformation and the physical view of data processing during debugging. For example, when two transformations `t1` and `t2` are applied to `x`, these are combined to `t2(t1(x))` in a single stage and the intermediate result of `t1(x)` is not viewable, as it is not materialized or stored.

Simulated Breakpoint. A simulated breakpoint enables a user to inspect intermediate results at a given breakpoint and resume the execution to create an illusion of a breakpoint, even though the program is still running on the cloud in the background. When a simulated breakpoint is hit, BIGDEBUG spawns a new process to record the transformation lineage of the breakpoint, while letting the executors continue processing the task. For example, in Figure 3.9, when a user sets a `breakpoint` after `flatmap`, program state `S2` is captured from the original workflow without affecting its execution. Therefore, setting a simulated breakpoint has almost zero overhead, as it only retains information to re-generate the program state from the latest materialization point, *i.e.,*

the last stage boundary before the simulated breakpoint, in this case `S1`.

When a user requests intermediate results from the simulated breakpoint, BIGDEBUG then recomputes the intermediate results and caches the results. If a user queries data between transformations such as `flatmap` and `map` in Figure 3.9 within the same stage, BIGDEBUG forces materialization of intermediate results by inserting a `breakpoint` and `watchpoint` (described in Section 3.4.2) API call on the RDD object to collect the intermediate results.

Resume and Step Over. When a user enters a `resume` command using BIGDEBUG's user interface, BIGDEBUG will automatically jump to the original workflow running in the background. This procedure improves the overall throughput of the distributed processing. When a user enters a `step over` command to investigate the state after the next transformation in the UI, BIGDEBUG replays the execution to the next instruction only from the latest materialization point. This feature differentiates BIGDEBUG from an existing replay debugger such as Arthur [50], which restarts a job from the beginning. In Figure 3.9, when a user selects `step over`, a new workflow will start from the nearest possible materialization point, in this case, `S1`. BIGDEBUG uses the materialized state `S1` and executes later operations while capturing `S3` on the go.

Realtime Code Fix. 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. Terminating a job at a later stage will waste all computations before. Because running tasks on cloud costs lots of money and even days to process billions of records, we hypothesize that developers are less likely to terminate the program after inspecting it at the breakpoint.

To save the cost of re-run, BIGDEBUG allows a user to replace any code in the succeeding RDDs after the breakpoint. If a user wants to modify code, BIGDEBUG applies the fix from the last materialization point rather than the beginning of the program to reuse previously computed results. Assuming that a breakpoint is in place, a user submits a new function (*i.e.,* a data transformation operator) at the driver. The function is then compiled using Scala's NSC [5] library and shipped to each worker to override the call to the original function, when the respective RDD is executed. Suppose a user sets a breakpoint after `flatmap` and the program is paused in Figure 3.9. A user

can replace the function in the `map` transformation from `((word => (word,1))` to {`word => if (word!=null) (word,1); else (word,0);`}. When a user resumes after the fix at `S2`, a new workflow will start from `S1` and later RDDs including modified ones will be computed until the end of the workflow is reached and the background job of the original workflow is terminated. BIGDEBUG checks whether the supplied function has the same type as the original function through static type checking. Therefore, a user cannot provide a code fix that breaks the integrity of the used data type. When a user replaces function `f` with a new function `g`, BIGDEBUG applies `g` to all records from the last materialization point to ensure consistency.

3.4.2 On-Demand Watchpoint with Guard

Similar to watching a variable in a traditional debugger like `gdb`, BIGDEBUG provides 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, as all worker nodes must transfer the intermediate results back to the driver node. To overcome these challenges, BIGDEBUG provides an **on-demand watchpoint** with a **guard** closure function. A user can provide a guard to query a subset of data matching the guard. For example, `(r=>r>4)` is an anonymous function that takes `r` as input and returns true, if r is greater than 4, and `rdd.watchpoint(r=>r>4)` sets a watchpoint to retrieve all records greater than 4. BIGDEBUG automatically compiles such user-provided guard and distributes it to worker nodes to retrieve the matching data.

On-Demand Batch Transfer. To reduce communication overhead, BIGDEBUG batches intermediate data and sends them to the driver when needed. If no request is made for the watchpointed data from the user, it will be kept at the workers until the end of the stage or the next breakpoint in the same stage, if there is any. When the computation passes the end of the stage, the remaining filtered data are flushed, so that there are enough memory available for other Spark operations.

Dynamic Guard Modification. A user can modify a guard function to narrow down the scope of captured data. This feature is called a *dynamic guard*, as the function can be refined iteratively

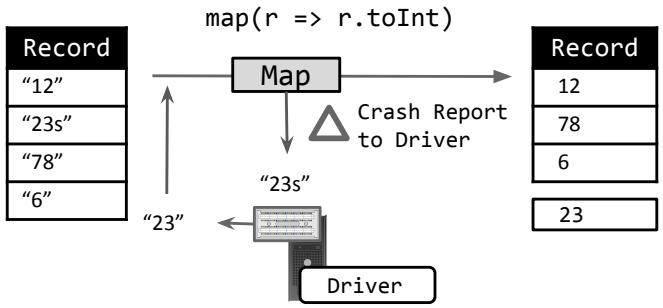


Figure 3.10: An intermediate record "23s" at the `map` transformation causes a crash. BIGDEBUG reports the crash culprit, "23s" to the user and a user supplies the corrected record, "23".

while the Spark job is executing. A watchpoint guard is built on top of Scala and Java and is written like a normal Spark program. For example, an expression `(value._1.length > 50 && value._2 == 1)` filters values where the length of first element in tuple is greater than 50 and the value of second element in tuple is 1. When a user updates a guard during a Spark job, BIGDEBUG uses Scala's NSC library [5] to compile the guard at the driver node and ships it to all workers. Individual workers then load the new guard at each executor. Using a dynamic guard, a user can tune the amount of data being transferred and presented.

3.4.3 Crash Culprit and Remediation

DISC systems are limited in their ability to handle failures at runtime. In Spark, crashes cause the 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. BIGDEBUG leverages fine-grained tracing to be discussed in Section 3.4.4 to identify a crash-inducing input, not just a crash culprit record in the intermediate stage. While waiting for a user intervention, BIGDEBUG runs pending tasks continuously to utilize idle resources and to achieve high throughput.

Crash Culprit Determination. When a crash occurs at an executor, BIGDEBUG sends all the required information to the driver, so that the user can examine crash culprits and take actions as depicted in Figure 3.10. When a crash occurs, BIGDEBUG reports (1) a *crash culprit*—an intermediate record causing a crash (2) a stack trace, (3) a crashed RDD, and (4) the original input record

inducing a crash by leveraging backward tracing in Section 3.4.4.

Remediation. BIGDEBUG avoids the re-generation of prior stages by allowing a user to either correct the crashed record, skip the crash culprit, or supply a code fix to repair the crash culprit. A naive resolution approach is to pause the execution, report the crash culprit to the driver and wait until a resolution is provided from the user. This method has the disadvantage of putting the workers in the idle mode, reducing throughput. Therefore, BIGDEBUG provides the following two methods.

- **Lazy Repair of Records.** In case of a crash, the executor reports a crash culprit to the driver but continues processing the remaining pending records. Once the executor reaches the end of the task, it then waits for a corrected record from the user. This approach parallelizes the crash resolution without holding back the executor. If there are multiple crash culprits, BIGDEBUG accumulates the crashed records at the driver and lets all executors terminate, except the very last executor. The last executor on hold then processes the group of corrected records provided from the user, before the end of the stage. This method applies to the pre-shuffle stage only, because the record distribution must be consistent with existing record-to-worker mappings. This optimization of replacing crash-inducing records in batch improves performance.
- **Lazy Code Fix.** BIGDEBUG accumulates crash culprits at the driver and lets all executors continue processing the pending records. It then asks a user to supply a code fix to repair the crash culprits, *i.e.,* a new repair function to apply to the crash-culprits. Our assumption is that the new function extends the original function to clean the crash-inducing records and a user would like to see some results rather than nothing, because the current Spark does not provide any output to the user when a task crashes, even for successful inputs. If a user wants to apply a completely different function g to all records, she can use our *Realtime Code Fix* at simulated breakpoint instead. Similar to the above lazy repair of records, the last executor on hold applies the supplied function to the crash culprits in batch before the end of the stage.

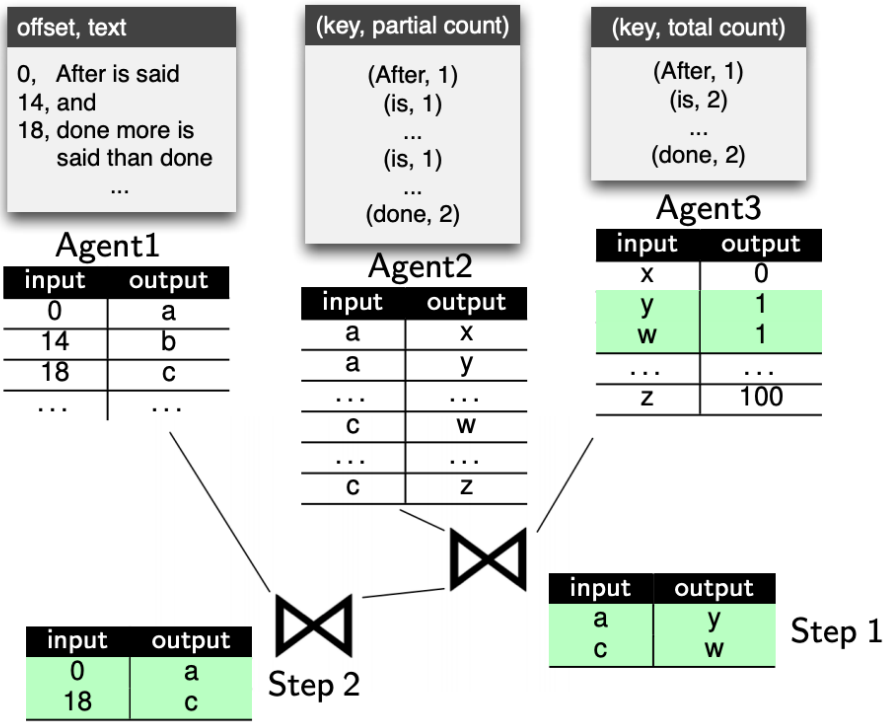


Figure 3.11: A logical trace plan that recursively joins data lineage tables, back to the input lines

3.4.4 Forward and Backward Tracing

BIGDEBUG supports fine-grained tracing of individual records by invoking a data provenance query on the fly. The *data provenance* problem in the database community refers to identifying the origin of final (or intermediate) output. Data provenance support for DISC systems is challenging, because operators such as `aggregation`, `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 worker nodes.

BIGDEBUG uses data provenance capability implemented through an extension of Spark's RDD abstraction (called *LineageRDD*) that leverages Spark's built-in fault tolerance and data management infrastructure [82]. The LineageRDD abstraction provides programmers with data provenance query capabilities. Provenance data is captured at the record level granularity, by tagging records with identifiers and associating output record identifiers with the relevant input record identifier, for a given transformation. From any given RDD, a Spark programmer can obtain a

LineageRDD reference and use it to perform *data tracing—i.e.,* the ability to transition backward (or forward) in the Spark program dataflow, at the record level.

BIGDEBUG instruments submitted Spark programs with *tracing agents* that wrap transformations at stage boundaries. These agents implement the LineageRDD abstraction and have two responsibilities: (1) tag input and output records with unique identifiers for a given transformation and (2) store the associations between input and output record identifiers as a data provenance table in Spark's native storage system. For example, Figure 3.11 shows intermediate results and a corresponding data provenance table for each agent. For example, the first line at offset 0 "After is said" is assigned with a unique output identifier, `a`. As this line is split into multiple records such as (After, 1) and (is, 1), unique output identifiers such as `x` and `y` are assigned to the corresponding key and partial count pairs. BIGDEBUG utilizes specific tracing agents based on the type of transformation *e.g.,* data ingested from Hadoop Distributed File System (HDFS) and Amazon S3, and all native Spark transformations; agents are pluggable, making it easy to support other input data storage environments *e.g.,* Cassandra, HBase, and RDBMS.

Once the provenance data is recorded, tracing queries can be issued using the lineage-related methods of the API in Figure 3.7. The `goBackAll` and `goNextAll` methods are used to compute the full trace backward and forward respectively. That is, given some result record(s), `goBackAll` returns all initial input records (*e.g.,* in HDFS) that contributed in the generation of the result record(s); `goNextAll` returns all the final result records that a starting input record(s) contributed to in a transformation series. A single step backward or forward is supported by the `goBack` and `goNext` respectively. At any given point in the trace, the user can interactively issue a native Spark `collect` action to view the raw data referenced by the LineageRDD.

When a tracing query is issued, BIGDEBUG logically reconstructs the path connecting input to output records by recursively joining the provenance tables generated by the tracing agents, as shown in Figure 3.11 using the word count example. After executing a word count job, a user may want to perform a full backward tracing from the output value $(\mathsf{is}, 2)$. BIGDEBUG does this by first retrieving the identifier for output $(\mathsf{is}, 2)$ from the output provenance table. Figure 3.11

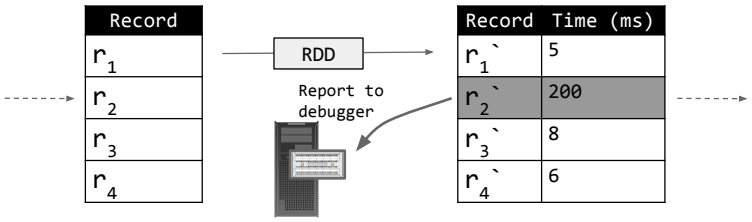


Figure 3.12: When latency monitoring is enabled, straggler records are reported to the user.

shows that identifier to be equal to 1, and the corresponding mappings have two input identifiers y and w. Tracing proceeds by (recursively) joining the provenance tables, at neighboring capture agents, along the output and input values. $\bowtie$ represents a join operation of provenance tables. For instance, the join of Agent 3 and Agent 2 produces the Step 1 trace result. Subsequently, joining Step 1 with Agent 1 produces the Step 2 trace result, which is the final trace result referencing HDFS input records at offsets 0 and 18, both containing the word "is".

Supporting data provenance, while logically simple, is difficult to achieve in a DISC environment such as Spark because the size of input-output identifier mappings is as large as all intermediate results, and data provenance tables are physically distributed across worker nodes. For these reasons, Spark's internal storage service is used for storing provenance data, and a distributed join implementation uses partition information embedded into the record identifiers to optimize the shuffle step. Implementing an *optimized distributed join* of data provenance tables in Spark is the subject of another paper [82], which details the advantage of an optimized distributed join in Spark over storing data provenance tables in external storage services (*e.g.,* HDFS, MySQL).

3.4.5 Fine-Grained Latency Alert

In big data processing, it is important to identify which records are causing delay. Spark reports a running time only at the level of tasks, making it difficult to identify individual *straggler records*—records responsible for slow processing. To localize performance anomalies at the record level, BIGDEBUG wraps each operator with a latency monitor. For each record at each transformation, BIGDEBUG computes the time taken to process each record, keeps track of a moving average,

and sends to the monitor, if the time is greater than k standard deviations above the moving average where default k is 2. Figure 3.12 shows an example of how straggler records are tagged and reported to the debugger. As we show in Section 3.6, record-level latency alert poses the highest overhead among BIGDEBUG's primitives due to the cost of taking a timestamp for processing each record and computing the moving average among millions of records per executor.

```
1  val log = "s3n://xcr:wJY@ws/logs/enroll.log"
2  val text_file = spark.textFile(log)
3  val avg = text_file
4       .map(line => (line.split()[2] , line.split()[3].toInt) )
5       .groupByKey()
6       .map(v => (v._1 , average(v._2)) )
7       .collect()
```

Figure 3.13: College student data analysis program in Scala

3.5 Tool Interfaces

In this section, we will walk through the tool features of BIGDEBUG with focus on tool demonstration. Suppose Alice wants to process all US college student data. Because of the dataset massive size, she cannot store and analyze the data in a single machine. Suppose that she intends to compute the average age of all college students in each year (freshman, sophomore, junior, and senior) using the program of Figure 3.13. `1 Timothy Sophomore 21` is the format of a sample input record.

She starts by loading the US college student data from an Amazon S3 storage into the cluster (line 2). She the parses the data into appropriate key-value date types, where a key is the status category for a student and the value is the age of that student (line 4). Records are then grouped with respect to the key, and the average for each category is computed (lines 5 and 6). Finally, at line 7 she executes the job and requests the result to be sent to the driver.

Simulated Breakpoint and Guarded Watchpoint. Suppose that Alice wants to inspect the program state at line 3. She can insert a simulated breakpoint using BIGDEBUG's API *i.e.,* `simulatedBreakpoint(r => !COLLEGEYEAR.contains(r.split()[2]))` with the guard predicate indicating that the second field is not one of the pre-defined college years. While the Spark

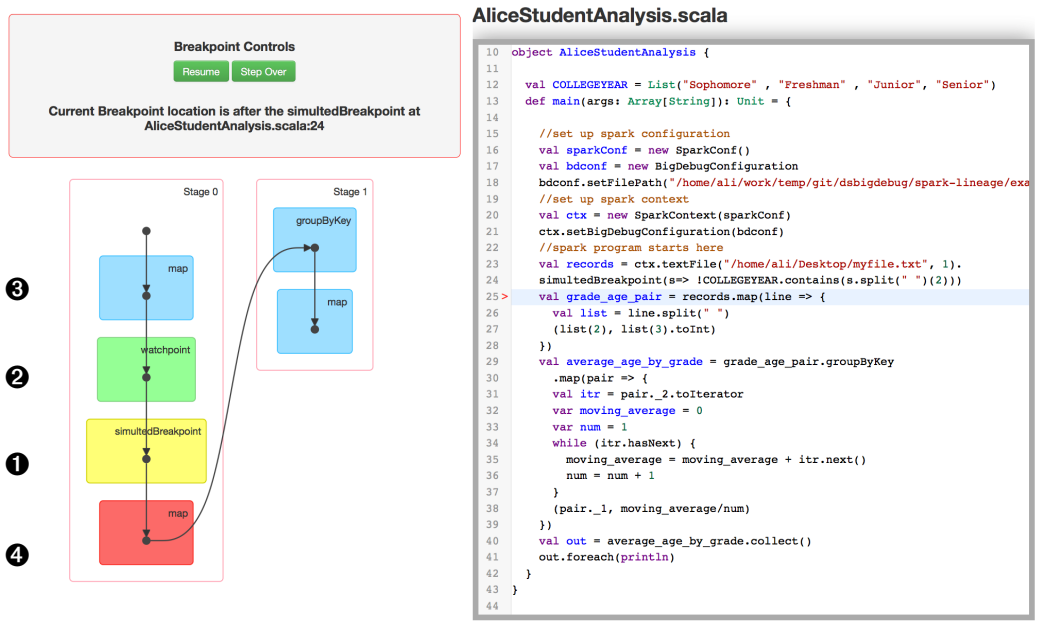


Figure 3.14: BIGDEBUG extends Spark's user interface to provide runtime debugging features

program instrumented with breakpoints is running on the cluster, Alice can use a web-based debugger interface by connecting to a configured port. Using this interface, she can view the DAG of the data flow program. On the left hand side of Figure 3.14, the yellow node (❶) in the DAG represents a breakpoint. Alice can use the code editor window on the right hand side to see the Spark program in execution. Statements with a breakpoint are tagged using a red arrow.

She can click on the green node (❷) on the DAG, which redirects her to a new web page, where intermediate records are displayed. When she requests to view the internal program state, the captured records from the guarded watchpoint are transferred to the driver node and displayed as shown in Figure 3.15. Upon viewing the intermediate records at the breakpoint, Alice discovers that some records use number 2 instead of Sophomore to indicate the status year:

```
1   Timothy   2   21
```

.

Realtime Code Fix. From the outlier records, Alice immediately learns that her program should

Figure 3.15: A user can edit the guard predicate using an editor.

handle records with a status year written in numbers. To apply realtime code fix, Alice can click on the corresponding transformation (❸) marked in blue in the DAG. She can then insert a new user-defined function to replace the old one using the related code editor provided by the BIGDEBUG UI. The code fix can now handle status year both in number and string formats. In addition to a realtime code fix feature, Alice can use *resume* and *step over* commands. These control commands are available in BIGDEBUG's UI.

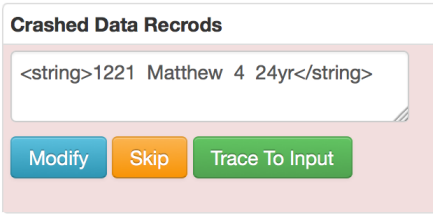


Figure 3.16: Options provided by the crash remediation UI

Crash Culprit Remediation. Suppose that, after several hours of computation, a runtime exception occurs during the data processing. BIGDEBUG alerts Alice on the intermediate record responsible for the crash. These alerts turn the corresponding transformation node of the DAG to be red (❹ in our example workflow of Figure 3.14) and highlight the corresponding code line in the main editor window to be red as well. When Alice clicks on the red node (❹) in the DAG, she is redirected to the crash culprit page of Figure 3.16. This page contains the following crash culprit record: `1221 Matthew 4 24yr`. Alice may skip or modify the crash inducing intermediate record directly. Figure 3.16 shows the options provided on the UI to perform these remediation operations on the crash-inducing records.

Forward and Backward Tracing. During crash remediation, Alice can invoke forward and backward tracing features at runtime to find the original input records responsible for the crash. On the crash culprit UI, Alice can invoke the backward tracing query by pressing the *trace to input*

button. BIGDEBUG performs backward tracing in a new process to trace crash-inducing records in the original input data.

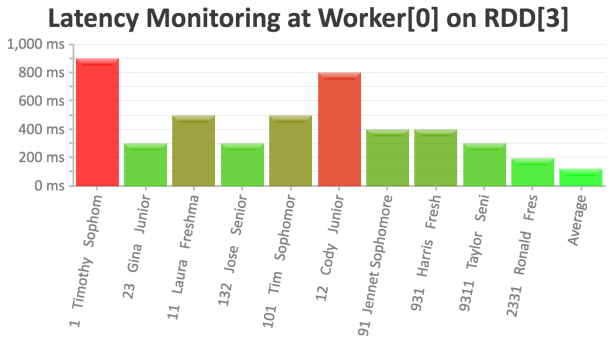


Figure 3.17: Top most straggling records are visualized in bar chart showing the delays relative to average

Fine-Grained Latency Monitoring. Alice can select a latency-enabled RDD from a drop-down menu to visualize straggler records (*i.e.,* delay-causing records) in a streaming fashion. Figure 3.17 depicts a set of stragglers that Alice sees when enabling latency monitoring for RDD 3.

3.6 Evaluation

We evaluate BIGDEBUG's (1) scalability, (2) performance overhead, (3) localizability improvement in determining crash-inducing records, and (4) time saving w.r.t to an existing replay debugger. The main purpose of our evaluation is to investigate whether BIGDEBUG keeps performance similar to the original Spark and retains its scalability, while supporting interactive debugging.

- How does BIGDEBUG scale to massive data?
- What is the performance overhead of instrumentation and additional communication for debugging primitives?
- How much localizability improvement does BIGDEBUG achieve by leveraging fine-grained, record level tracing?
- How much time saving does BIGDEBUG provide through its runtime crash remediation, in comparison to an existing replay debugger?

We use a cluster consisting of sixteen i7-4770 machines, each running at 3.40GHz and equipped

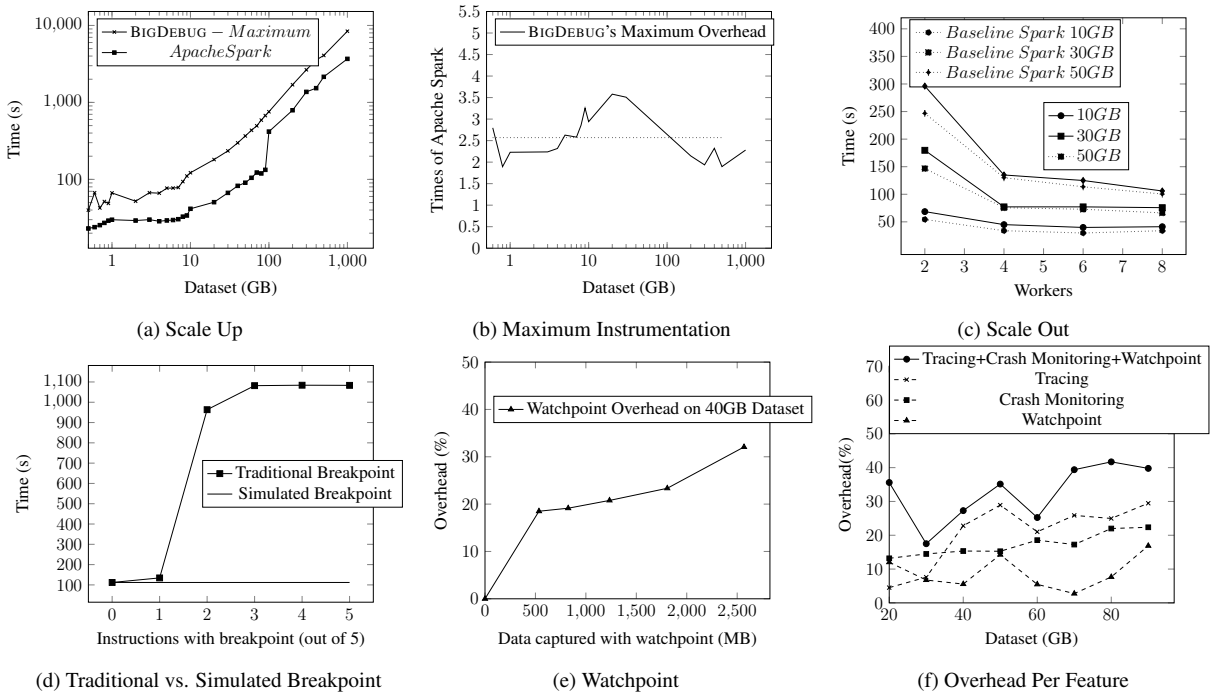
Benchmark	Dataset(GB)	Overhead	
		Max	w/o Latency
PigMix L1	1, 10, 50, 100, 150, 200	1.38X	1.29X
Grep	20, 30, 40, . . . 90	1.76X	1.07X
Word Count	0.5 to 1000 (increment with a log scale)	2.5X	1.34X

Table 3.1: Performance evaluation on subject programs

with 4 cores (2 hyper-threads per core), 32GB of RAM, and 1TB of disk capacity. The operating system is a 64bit Ubuntu 12.04. The datasets are all stored on HDFS version 1.0.4 with a replication factor of 1—one copy of each dataset across the cluster. We compare BIGDEBUG's performance against the baseline Spark, version 1.2.1. The level of parallelism was set at two tasks per core. This configuration allows us to run up to 120 tasks simultaneously.

We use three Spark programs for our performance experiments: *WordCount*, *Grep*, and *PigMix query L1*. *WordCount* computes the number of word occurrences grouped by unique words. *WordCount* comprises of 4 transformations, which are divided into 2 stages: the first stage loads the input file (`textfile`), splits each line into a bag of words (`flatmap`), creates a tuple of (word, 1) (`map`), and the second stage reduces the key-value pairs using each word as the key (`reduceByKey`). *Grep* finds the lines in the input datasets that contain a queried string. *Grep* comprises of only 1 stage with 2 transformations : `textfile` reads the input file line by line and `filter` applies a predicate to see if the line contains a substring. PigMix's latency query L1 is a performance benchmark for DISC systems in which an unstructured data is transformed and analyzed. L1 comprises of 2 stages: the first stage contains `textfile`, 2 `maps`, `flatmap` followed by 2 `maps` and the second stage has `reduceByKey` followed by a `map`.

For our performance experiment, we vary input size from 500MB to 1TB by using an unstructured data made up of `Zipf` distribution over a vocabulary of 8000 terms. All runs are repeated 10 times. We compute a *trimmed mean* by removing the shortest 2 runs and the longest 2 runs, because the running time of big data applications depends on various factors such as a warm up of HDFS cache, garbage collection, and network I/O. In big data systems, a variation of 5% is considered noise, because of these factors.



(a) Scale Up

(b) Maximum Instrumentation

(c) Scale Out

(d) Traditional vs. Simulated Breakpoint

(e) Watchpoint

(f) Overhead Per Feature

Figure 3.18: BIGDEBUG's scalability and overhead

3.6.1 Scalability

We perform experiments to show that BIGDEBUG scales to massive data, similar to Spark. More specifically, we assess the *scale up* property—BIGDEBUG can ingest and process massive data and its running time increases in proportion to the increase in data size. We assess the *scale out* property—as the number of worker nodes (the degree of parallelization) increases, the running time decreases.

Scaling Up. Figure 3.18a shows the scale-up experiment while varying the data size from a few gigabytes to one terabyte. In this experiment, BIGDEBUG is used with the maximum instrumentation where breakpoints and watchpoints are set in every line and latency monitoring, crash remediation, and data provenance are enabled for every record at every transformation. The running time of BIGDEBUG grows steadily in proportion to the running time of Spark. As the data size increases, BIGDEBUG's running time also increases, because large input data requires the scheduler to create more tasks, assign the tasks to workers, and coordinate distributed execution to improve data

locality, throughput, and HDFS caching.

With such maximum instrumentation, BIGDEBUG scales well to massive data (1TB). Figure 3.18b shows the overhead in *WordCount*. BIGDEBUG takes 2.5X longer on average in comparison to the baseline Spark. This 2.5X overhead is a very conservative upper bound, as a developer may not need to monitor the latency of each record at every transformation and may not need to set breakpoints on every operator. When disabling the most expensive feature, record-level latency monitoring, BIGDEBUG poses an average overhead of 34% only. In *Grep* and *L1*, the overheads are 1.76X and 1.38X with the maximum instrumentation and 7% and 29% respectively when latency monitoring is disabled (see Table 3.1).

Scaling Out. The property of scaling out is essential for any DISC system. We change the number of worker nodes (essentially the total number of cores) on a fixed size dataset and record a running time for both BIGDEBUG and the baseline Spark. As the number of cores increases, the running time should decrease. Figure 3.18c shows that BIGDEBUG does not affect the scale-out property of the baseline Spark. When there are too many workers, it makes it hard for the Spark scheduler to retain locality, and BIGDEBUG also ends up moving data around at stage boundaries between workers.

3.6.2 Overhead

We measure the overhead as the increase in the running time of BIGDEBUG w.r.t the baseline Spark. The performance overhead comes from instrumentation and communication between the driver and the workers and data transfer to carry the requested debug information to the user. BIGDEBUG works at the record level. Therefore, the overhead increases in proportion to the number of records in each stage. For example, when the *WordCount* program splits input lines into a bag of words at the `flatmap` transformation, the overhead increases.

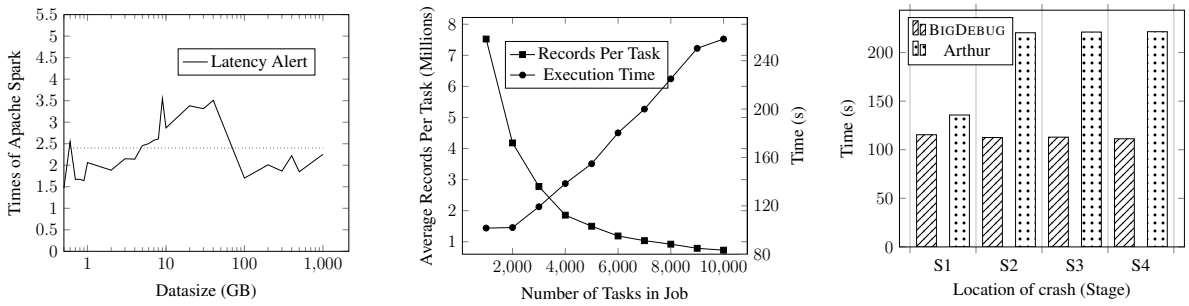
Simulated Breakpoint. BIGDEBUG offers almost 0% overhead to pause and instantly resume. This overhead is the same across all subject programs because instantly resuming simulated breakpoint does not involve any additional instrumentation. On the other hand, a traditional breakpoint

adds overhead because it must pause and cache the intermediate state entirely. Figure 3.18d shows the comparison results between simulated breakpoints and traditional breakpoints, while setting a breakpoint at different transformations. Traditional breakpoints incur orders of magnitude higher overhead, since they materialize all intermediate results regardless of whether a user requests them or not. For example, a traditional breakpoint's overhead rises sharply after a `flatmap` transformation that emits a large number of unaggregated records.

On-Demand Watchpoint. We place a watchpoint between two instructions such that a custom guard reports only a certain percentage of intermediate results, *e.g.,* 2%, 5% , 10% etc. Once captured, the intermediate data is sent to the driver for a user to inspect. In Figure 3.18e, the x-axis shows the amount of captured data ranging from 500MB to 3GB, when the total intermediate data size is 40GB. When 500MB of data is transferred to the user at the watchpoint, it incurs only 18% overhead. This is a very conservative set up, since the user is unlikely to read 500MB records at once. To isolate the overhead of setting watchpoints from the overhead of data transfer, we set a watchpoint at every transformation with a guard that always evaluates to false. Among all subject program, *WordCount* poses the maximum overhead of 9% on average, whereas the overhead for *Grep* and *L1* is 5% and 3% respectively.

Crash Monitoring. We enable crash monitoring for every transformation and vary the data size from 20GB to 90 GB to measure overhead. See Figure 3.18f. Crash monitoring imposes 19% overhead in *L1*. *WordCount* and *Grep* incur a lower overhead of 18% and 4% respectively. This overhead comes from monitoring every record transformation for any kind of a failure, and checking if there are any pending crashed records to be repaired at the end of local execution in a task.

Backward and Forward Tracing. We enable record-level data provenance capturing for every transformation and vary the data size from 20GB to 90GB. See Figure 3.18f. The tracing primitive poses, on average, 24% overhead over the baseline Spark in *L1*. *WordCount* and *Grep* pose, on average, 22% and 5% overhead respectively. The majority of the overhead comes from the generation of the data provenance tables maintaining the associations between input and output record identifiers. The cost of storing each data provenance table into Spark's storage layer is small,



(a) Latency Alert Overhead (b) Localizability vs. Performance (c) Time Saving

Figure 3.19: Latency monitoring overhead, time saving through crash remediation, and localizability improvement

because it is performed asynchronously.

Fine-Grained Latency Alert. We enable record-level latency monitoring on every transformation. Record-level latency monitoring incurs the highest overhead among BIGDEBUG's features. Latency monitoring alone takes 2.4X longer than baseline Spark on data size varying from 500MB to 1TB (see Figure 3.19a) in *WordCount*. In *Grep* and *L1*, the overhead is 1.74X and 1.24X. The significant overhead comes from performing a statistical analysis for each record transformation. A timestamp is recorded before and after the transformation of each record to see if its latency lies within 2 standard deviations from the average. The overhead is also incurred by updating a moving average and a standard deviation. If we disable record-level latency monitoring from the maximum instrumentation setting, the overhead decreases from 150% (*i.e.,* 2.5X of baseline Spark) to 34% (*i.e.,* 1.34X of baseline Spark) in the case of the *WordCount* program.

3.6.3 Crash Localizability Improvement

Spark reports failures at the level of tasks, while BIGDEBUG reports specific failure-inducing inputs. BIGDEBUG also detects specific straggling records, while Spark only reports a straggler task causing delay. By construction, our record-level tracing approach has 100% accuracy with zero false positive because it leverages *data provenance* to identify all inputs contributing to a failure. Delta Debugging [147] can further isolate this combination of multiple failure-inducing inputs, but will require a larger number of runs, as opposed to BIGDEBUG that requires only a single run.

Therefore, we measure the improvement in localizing failed (or delayed) records in comparison to the baseline Spark. When debugging a program using Spark, a developer may want to increase the number of tasks to improve fault localizability at the cost of running time. Note that the running time increases as you increase the number of tasks, since more resources are used for communication and coordination among distributed worker nodes. To quantify this, we vary the number of tasks and measure the average number of records per task and the total running time. See Figure 3.19b. When configuring Spark with 1000 tasks and 60GB dataset, each task handles 7.52 million records on average. If a single record among 7.52 million records crashes the task, it is impossible for a human being to identify a crash culprit. To improve fault localizability, if a developer increases the number of tasks from 1000 to 10,000 to reduce the number of records assigned to each task, each task still handles 0.72 million records, and additional communication and coordination increases the total running time by 2.5 times. As Figure 3.18f shows, BIGDEBUG incurs less than 19% overhead for reporting crash culprits and takes only 2.4X time for latency alert, while improving fault localizability by orders of millions.

3.6.4 Time Saving through Crash Remediation

When a task crashes, the current Spark does not provide any output to the user, even for successful inputs and terminates the task immediately. On the other hand, BIGDEBUG allows a user to remove or modify a crash culprit at runtime to avoid termination. Therefore, BIGDEBUG avoids additional runs when a user tries to remove crash-inducing records from the original input. For the same scenario, using a *post-hoc instrumentation replay* debugger Arthur [50] requires at least three runs. In the first run, a program crashes and Spark reports failed task IDs. In the second run, a user must write a custom post-hoc instrumentation query (a new data flow graph) with those failed task IDs and run the query to recompute the intermediate results for the failed tasks. In the third run, a user removes the crash-inducing records and re-runs the job again. Crashes in later stages result in more redundant work in the second and third runs and hence more time for completion. When a crash occurs at the 9th stage of a 10 stage job, Arthur must recompute the first 9 stages twice,

while BIGDEBUG avoids such re-computation completely by allowing a user to resolve crashes in the first run.

Figure 3.19c shows our experiment result on time saving. We compare BIGDEBUG's time saving with a *post-hoc* instrumentation replay debugger like Arthur. We conservatively estimate Arthur's performance by running the original Spark for its first and third runs. We measure time saving by dividing the additional time required by Arthur by BIGDEBUG's completion time. We seed crashes in different transformation locations by updating an original program to throw an exception at a given stage, because the magnitude of time saving depends on which stage a crash occurs. For example, we replace the map function {`word => (word,1)`} with {`word=> if(word == "Crash") crash(); (word,1)`} where `crash()` always throw a `NullPointerException`. S1 is a program where crashes are seeded in the first stage, S2 is a program where crashes are seeded in the second stage, etc. BIGDEBUG saves the execution time by 80% on average and reaches up to 100% after S2. In the experiment, the most time consuming stage is S2 and a crash in S2 or later saves a large amount of time.

3.7 Discussion

Through our extensive evaluation, we validate our sub-hypothesis (**SH1**) that interactive debugging primitives for big data analytics can reduce the debugging time significantly without compromising the throughput of big data applications. As of any other interactive debugger, BIGDEBUG relies on user judgment to set breakpoints and watchpoints at the right place in the program. Furthermore, a fault in big data application may surface several transformations later into the execution in the form of failure or incorrect record. BIGDEBUG performs backward tracing to isolate the crash-inducing input records; however, such input may still be in the order of millions, which are infeasible to inspect manually. This motivates us to explore the possibility of fully automating the root-cause analysis when a user observes a suspicious intermediate program state at a breakpoint or an incorrect final output. In the next chapter, we investigate our sub-hypothesis (**SH 2**) and ask how we can provide a tool-assisted automated fault localization in big data analytics through which a user

can isolate the precise fault-inducing input records and diagnose the root cause of an error, failure, or outlier.

CHAPTER 4

Automated Debugging for Big Data Analytics

The previous chapter aims to facilitate the real time debugging with a wide variety of interactive debugging primitives suited for big data applications that run on a cloud computing environment. However, we still rely on developer's judgement and perception to use debugging primitives such as breakpoint and watchpoint at the correct location to isolate an intermediate failure. To reduce the burden of manual debugging, we ask the following research question: *how can we automatically isolate the root cause of a crash, failure, or suspicious?* In this chapter, we aim to automate the process of fault isolation and investigate our sub-hypothesis **SH2**: *A combination of data provenance technique from databases and a test-based fault-isolation technique from software engineering can help developers pinpoint the minimal subset of failure-inducing inputs.*

4.1 Introduction

Due to the heterogeneity of data sources in big data analytics, developers often deal with program errors and incorrect inputs *e.g.,* unclean data or making the wrong assumptions about the data [93]. The first step in debugging such issues is *simplification*—eliminating all irrelevant details and producing a minimum example that still produces the failure [147].

We present BIGSIFT, a new fault localization approach that combines insights from both software engineering and database literature to bring delta debugging closer to a reality in DISC environments. Given a test function, BIGSIFT automatically finds a minimum set of fault-inducing input records responsible for a faulty output. We re-define data provenance [82] 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 Delta Debugging.

Our evaluation shows that BIGSIFT finds the most concise subset of fault-inducing input where data provenance stops at identifying failure-inducing records at the size of up to $\sim 10^3$ to 10^7 records. In comparison to using Delta Debugging, BIGSIFT reduces the fault localization time (as much as $66\times$) by pruning out input records that are not relevant to faulty outputs. More surprisingly, the total debugging time taken by BIGSIFT is 62% less than the original job running time per single faulty output.

The rest of the chapter is organized as follows. Section 4.2 describes a motivating example. Section 4.3 describes the design and implementation of BIGSIFT. Section 4.4 demonstrates BIGSIFT on a big data application. Section 4.5 describes evaluation settings and the corresponding results. Section 4.6 concludes the chapter and discusses future research direction.

4.2 Motivating Example

This section discusses a motivating example to elucidate the challenges of debugging DISC system workloads and the limitations of DD and DP approaches in addressing this challenge.

Alice writes a Spark program to process a large dataset that contains weather telemetry data of the U.S. over several years. She wants to compute the delta between the minimum and the maximum snowfall measurement in each state for (1) each day of any year and (2) for each year. Data records are in CSV format: for example, the following sample record indicates that on January 1st of Year 1992, in the 99504 zip code (Anchorage, AK) area, there was 1 foot of snowfall: `99504 , 01/01/1992 , 1ft`.

To analyze the data, Alice develops the Spark program shown in Figure 4.1. She starts projecting each base record into two records (lines 3-17); the first representing the state, the date (`mm/dd`), and its snowfall measurement, and the second representing the state, the year (`yyyy`), and its snowfall measurement. She normalizes the snowfall measurements using the function `convertToMm`

```
1  val log = "s3n://xcr:wJY@ws/logs/weather.log"
2  val split = sc.textFile(log).flatMap{s =>
3    val tokens = s.split(",")
4    // finds the state for a zipcode
5    var state = zipToState(tokens(0))
6    var date = tokens(1)
7    // gets snow value and converts it into millimeter
8    val snow = convertToMm(tokens(2))
9    //gets year
10   val year = date.substring(date.lastIndexOf("/"))
11   // gets month / date
12   val monthdate= date.substring(0,date.lastIndexOf("/")-1)
13     List[((String , String) , Float)](
14         ((state , monthdate) , snow) ,
15         ((state , year) , snow)
16       )
17 }
18 val deltaSnow = split.groupByKey().map{ s =>
19  val delta = s._2.max - s._2.min
20  (s._1 , delta)
21 }
22 deltaSnow.saveAsTextFile("hdfs://s3-92:9010/output/")
23 def convertToMm(s: String): Float = {
24  val unit = s.substring(s.length - 2)
25  val v = s.substring(0, s.length - 2).toFloat
26  unit match {
27   case "mm" => return v
28   case _ => return v * 304.8f
29   }
30 }
```

Figure 4.1: Alice's program that identifies, for each state in the US, the delta between the minimum and the maximum snowfall reading for each day of any year and for any particular year. Measurements can be either in millimeters or in feet. The conversion function is described at line 23.

(described at line 23), which converts any units of feet to millimeters, based on an assumption she makes about the data. She also uses a function `zipToState` (line 5) to find the name of the state where an input zip code resides.

Next, she groups the key value pairs using a `groupByKey` operator in line 18, yielding records that are grouped in two ways (a) by state and day and (b) by state and year. At lines 18-21, Alice finds the delta between the maximum and the minimum snowfall measurements for each group and saves the final results to an HDFS directory. A snippet of the execution is shown in Figure 4.2(a), where, on January 1st, the snowfall level delta in Alaska (AK) is `21251` millimeters, and in year `1992`, the snowfall level delta in Alaska is `274.3` millimeters.

```
1  def test(key:String, delta: Float) : Boolean = {
2    delta < 6000
3  }
```

Figure 4.3: Test function checking the validity of each output record—all snowfall deltas greater than 6000 millimeter are suspicious or incorrect.

Suppose that Alice writes a test function to check the validity of her output records, as seen in

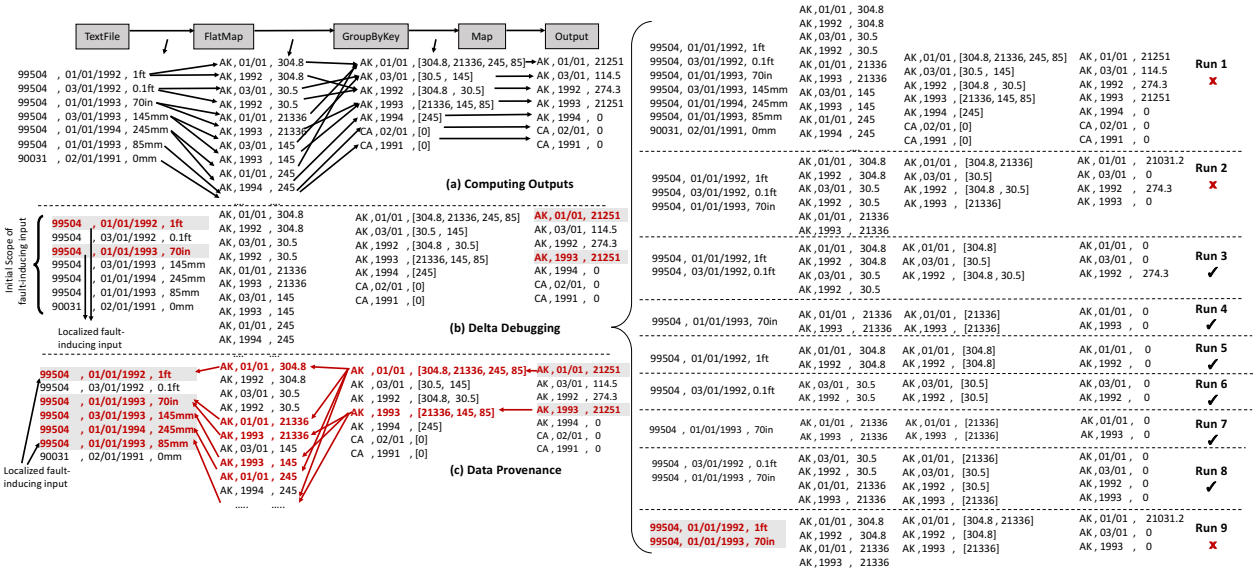


Figure 4.2: (a) shows how intermediate and final results are constructed using the transformations in Alice's program. In (b), delta debugging considers the initial scope of the fault-inducing input to span a complete dataset but it eventually finds the precise fault-inducing input. In (c), data provenance over-approximates the set of fault inducing input records because of the `groupByKey` operation in the initial program.

Figure 4.3. In her test function, she assumes that any delta snowfall level greater than 6000 millimeters (6 meters) is extremely suspicious, such as the delta snowfall of `21251`. However, once such outlier snowfall levels are identified, it is challenging for Alice to derive (by inspection) the precise set of input records leading to such faulty outputs, because the program involves computing both *min* and *max* over a unit conversion, making it hard to write a data cleaning filter upfront, as snowfall levels could vary greatly.

The goal of BIGSIFT is to identify the precise input records leading to each faulty output record. In this case, the faulty output records are caused by an error in the unit conversion code, because the developer could not anticipate that the snowfall measurement could be reported in the unit of *inches* and the default case converts the unit in feet to millimeters (line 28 in Figure 4.1). Therefore, the snowfall record `99504 , 01/01/1993 , 70in` is interpreted in the unit of *feet*, leading to an extremely high level of snowfall, like `21366` mm, after the conversion. In this case, BIGSIFT finds the minimum set of failure-inducing records: `99504 , 01/01/1992 , 1ft` and

`99504 , 01/01/1993 , 70in`, from which the unit measurements can be inspected, prompting a correction in the `convertToMm` function.

Limitations of Delta Debugging. *Delta Debugging (DD)* addresses the problem of isolating failure-inducing inputs by repetitively running a program with different sub-configurations of input. DD splits the original input into two halves using a binary search-like strategy and re-runs them. If one of the two halves fails, DD recursively applies the same procedure for only that failure-inducing input configuration. On the other hand, if both halves pass, DD tries different sub-configurations by mixing fine-grained sub-configurations with larger sub-configurations (computed as the complement from the current configuration). Under the assumption that a failure is *monotone*—where C is a super set of all input configurations, if a larger configuration c is successful, then any of its smaller sub-configurations c' does not fail, *i.e.,* $\forall c \subset C \ (\ test(c) = \checkmark \rightarrow \forall c' \subset c \ (test(c') \neq \text{✗}))$, DD returns a minimal failure-inducing configuration. The minimal failure-inducing configuration c_x means that removing any subset from c_x no longer fails: $\forall \delta_i \subset c_x, test(c_x - \{\delta_i\}) \neq \text{✗}$.

One limitation of delta debugging is that it is a black box procedure that does not consider the semantics of underlying data flow operators. In our running example, since the faulty output `AK, 01/01, 21251` is over state `AK` and date `01/01` only, we can easily conclude that the scope of failure-inducing input records should be limited to the records with date `01/01` and a zip code that exists in Alaska. However, delta debugging considers the entire input dataset (**Run 1**) as the initial scope of potential fault-inducing input, as it does not account for the semantics of the used data flow operators and keys.

Limitation of Data Provenance. *Data provenance (DP)* is a well-known technique in the database community for understanding the relationship between the input (or intermediate) records and the output records [19, 27, 47, 74]. For example, Titian is a data provenance tool for Apache Spark that allows users to perform forward and backward tracing of specific data records to understand the generation and consumption of data records [82]. If a user requests backward tracing for an output record, Titian identifies all the input records responsible for generating the output record

from a series of transformations. As such, DP could identify a subset of the dataset containing the failure-inducing inputs by backward tracing from the faulty output(s).

Assume now that Alice uses Titian to perform backward tracing of the faulty output record. Titian creates correspondences from the intermediate records `AK , 01/01 , 245` `AK , 01/01 , 85` `AK , 01/01 , 304.8` `AK , 01/01 , 21336` to a single output `AK , 01/01 , 21251`, as seen in Figure 4.2(c). Though only two of the input records (the minimum and the maximum value) contributed towards the final output across `groupByKey` and `map`, DP over-approximates, by a significant amount, the scope of fault-inducing records by simply assuming that all input records in the list to `groupByKey` contributed to the output record.

4.3 Approach

BIGSIFT implements a unique combination of data provenance (DP) and delta debugging (DD) to offer a toolkit for efficiently debugging DISC system workloads. It accepts a big data application, an input dataset, and a user-defined test function that distinguishes the faulty outputs from the correct ones. It then executes the application, and uses the test function to identify faulty output records. The debugging process is performed in three phases.

Phase 1 applies *test driven data provenance* (TP) 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* [131]. 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

Algorithm 1 BIGSIFT's algorithm

$local\_threshold$: an input size threshold on jobs for local computation
$test(c)$ runs the program on configuration c and checks whether it fails the test, $test(c_{✗})$ =✗ and $test(\emptyset)$ =✓
$testCombiners(t, I)$ filters the partial result that fails test t
$faults$: a minimum set of fault inducing input records
$split(c, n)$ splits the input c into n configurations

1: **if** $combinersForLastOperation$ **then** ▷ Phase I: Test Pushdown
2: $faulty\_output = testCombiners(test, input)$
3: **else**
4: $faulty\_output = testOutput(test, input)$
5: $C_L \leftarrow getLineage(faulty\_output)$ ▷ Phase I: Data Provenance
6: $C_L = SmallestJobFirst(C_L)$ ▷ Phase II: Smallest Job First
7: **while** !$C_L.isEmpty()$ **do**
8: **if** $|C_L| > 1$ **then**
9: $C_L, c_{INT} = overlap(C_L)$ ▷ Phase II: Trace Overlapping
10: $faults.push(ddmin_2(c_{INT}, 2))$
11: $faults.push(ddmin_2(C_L.pop(), 2))$
12: $faults.push(ddmin_2(C_L.pop(), 2))$
13: **else**
14: $faults.push(ddmin_2(C_L.pop(), 2))$
15: **return** $faults$
16: **function** $ddmin_2(c_{✗}, n)$ ▷ Phase III: Delta Debugging
17: $C \leftarrow split(c_{✗}, n)$
18: $(\Delta_i, testResult) = submitJob(C)$
19: **if** $testResult ==$ ✗ **then**
20: **return** $ddmin_2(\Delta_i, 2)$
21: **for** $\Delta_i \in C$ **do**
22: $C[i] = c_{✗} - \Delta_i$
23: $(\Delta_i, testResult) = submitJob(C)$
24: **if** $testResult ==$ ✗ **then**
25: **return** $ddmin_2(\Delta_i, max(n-1, 2))$
26: **if** $n < |c_{✗}|$ **then**
27: **return** $ddmin_2(c_{✗}, min(|c_{✗}|, 2n))$
28: **else**
29: **return** $c_{✗}$
30: **function** $submitJob(C)$
31: $testResult =$ ✓
32: **for** $\Delta_i \in C$ **do**
33: **if** $isTestMemoized(\Delta_i)$ **then**
34: $testResult = getTestResult(\Delta_i)$
35: **else** ▷ Phase III: Adaptive Scheduling
36: **if** $|\Delta_i| > local\_threshold$ **then**
37: $(\Delta_i, testResult) = runOnSpark(test, \Delta_i)$
38: **else**
39: $(\Delta_i, testResult) = runOnLocal(test, \Delta_i)$
40: $memoize(\Delta_i, testResult)$ ▷ Phase III: Test Memoization
41: **if** $testResult ==$ ✗ **then**
42: **return** $(\Delta_i, testResult)$
43: **return** $(\emptyset, testResult)$
44: **function** $overlap(C_L)$
45: $c_{INT} \leftarrow C_L(0) \cap C_L(1)$
46: **if** $test(C_{INT}) ==$ ✗ **then**
47: $C_L(0) = C_L(0) - c_{INT}$
48: $C_L(1) = C_L(1) - c_{INT}$
49: **return** (C_L, c_{INT})
50: **return** $(C_L, \emptyset)$

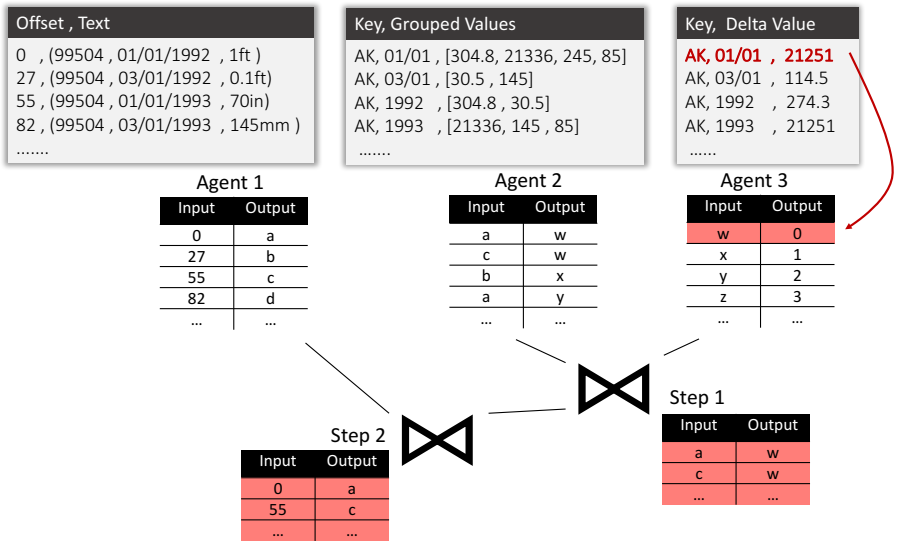


Figure 4.4: A logical trace plan that recursively joins data lineage tables back to the input lines.

test results of previously tried sub-configurations, when possible. Eventually, BIGSIFT outputs the smallest subset of input records responsible for the test failure of each faulty output. The debugging process of BIGSIFT is illustrated in Algorithm 1.

4.3.1 Phase I: Test Driven Data Provenance

In test-oracle based debugging such as DD, faulty outputs are distinguished from correct ones using a user-defined test function. Therefore, we could invoke a backward tracing query on each faulty output using a data provenance technique Titian to reduce the initial scope of fault-inducing inputs [82]. We describe how to use basic data provenance to identify an initial scope, and then how BIGSIFT extends it for test-oracle based debugging.

Data Provenance. When a Spark job is submitted, its workflow is generated in the form of a DAG. Titian takes in that DAG and inserts tracing agents in the workflow. These tracing agents modify the data records by attaching an identifier to each individual record. At every stage boundary, these ids are collected and added to an agent table that maintains mappings between the input and output records. When a tracing query is issued, Titian recursively joins the agent tables as shown in Figure 4.4, which illustrates a trace from output record `AK , 01/01 , 21251` (id `0`) to the records in the input file that derived it. Details on DAG instrumentation, distributed join of tracing

tables, and API usage can be found elsewhere [82]. For each faulty output, the corresponding fault-inducing input set from Titian is stored in a queue, C_L (line 5 in Algorithm 1).

Test Function Push Down. Spark applications comprise of hundreds to thousands of tasks running in parallel on different partitions. 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 to minimize network communication. Since Phase I 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.

Because faulty intermediate data records could have been already grouped together with non-faulty records from other partitions in an aggregation operation, we use the approach of pushing down a test-function to the earlier stage (*i.e.,* combiner) to isolate fault-inducing partitions. In the intermediate stage where a test function could be moved to, BIGSIFT then determines which partitions are no longer relevant to faulty outputs and therefore obviates the need of tracing non-faulty partitions further.

Specifically, in Apache Spark, certain aggregation operators (*e.g.,* `reduceByKey`) require a user to provide an *associative* and *commutative* function as an argument. For a test function applied to these operators, BIGSIFT can push-down the user-defined test function to partitions in the previous stage to test intermediate results (line 2 in Algorithm 1) if the following three conditions are met: (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, which is analogous to the monotonicity assumption of DD, meaning that an inclusion of a failure-inducing intermediate record(s) in the partition produces a test failure, when combined with other intermediate data from other partitions. If these three requirements are not met, BIGSIFT rollbacks to using basic data provenance. Therefore, the applicability of test-driven provenance optimization does not affect debugging accuracy. Rather, when these conditions are met, BIGSIFT can speed up debugging time.

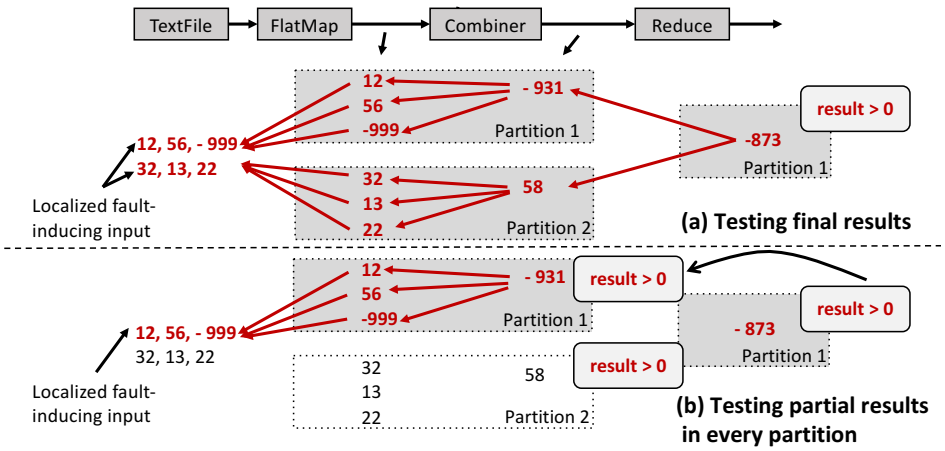


Figure 4.5: A decrease in the scope of potential fault-inducing input when the test function is pushed down. The workflow computes the sum of all the numbers in the input dataset.

```
1  sc.textFile(input)
2    .flatMap{s => s.split(",").map(r => r.toInt)}
3    .reduce( (a,b) => a+b )
4    .collect()
```

Figure 4.6: A Spark program that computes the sum of all the numbers in the input dataset.

When the three conditions are met, $f_1 \circ f_2$ could be checked at each partition before the shuffle stage as a *combiner*, identifying faulty partitions early. For the individual partitions failing this combiner test function, BIGSIFT restricts backward tracing search only on those faulty partitions, significantly reducing the scope of potential fault-inducing input records. On the other hand, if the monotonicity property is not satisfied (which can be verified by testing the final output), or none of the partitions fail the test function, BIGSIFT rolls back to the default case of backward tracing using basic data provenance.

Figure 4.5 contrasts data provenance without vs. with this push down optimization on the program in Figure 4.6. This program computes the sum of all numbers in the input dataset. It first splits each line in the input into a list of numbers using `flatmap` and then uses an *add* function as a UDF for the `reduce` operator. Suppose that a user-provided test function checks whether the final output is greater than zero. Spark automatically inserts a combiner by pushing the test function `result=>(result>0)` to check the partial results from the combiner. Figure 4.5(b) shows that BIGSIFT checks the intermediate results of Partition 1 (on which the test fails) and restricts its backward tracing to only this partition, resulting in a significantly smaller subset of records. On

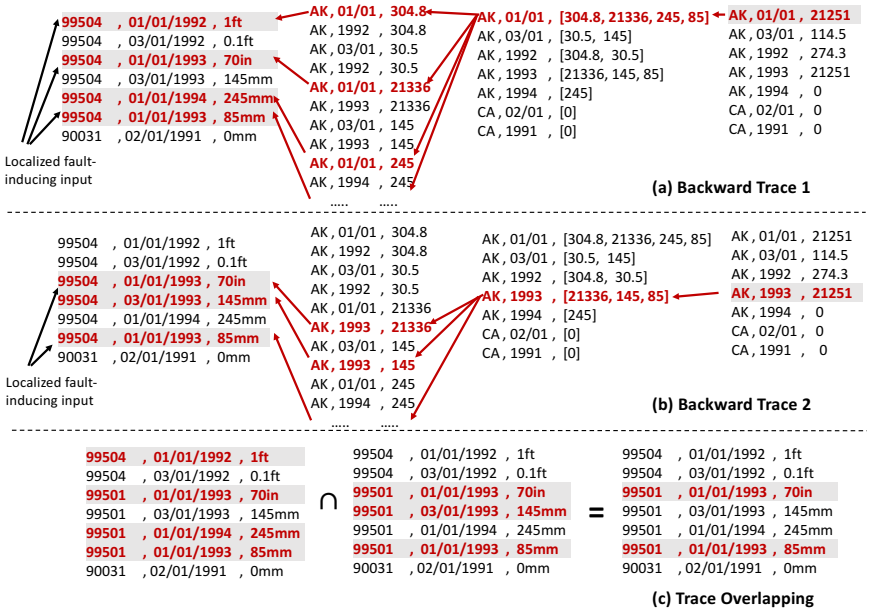


Figure 4.7: A decrease in fault-inducing input by overlapping backward traces of two faulty outputs emerging from the same fault-inducing input.

the other hand, Figure 4.5(a) shows that, without this optimization, all partitions are considered for backward tracing.

4.3.2 Phase II: Prioritizing Backward Traces

Phase II applies to the case when we have multiple faulty output records to explain. It takes the set of backward traces as input and employs two prioritization heuristics *i.e., trace overlapping* and *smallest job first*, based on the insight that multiple failure symptoms could be caused by the same set of inputs. BIGSIFT prioritizes backward traces to cover many faulty outputs within a time limit. When there is only one faulty output, Phase II is skipped.

Smallest Jobs First. Given multiple backtrace lineages from Phase I, BIGSIFT prioritizes the trace with the smallest number of potential fault-inducing input records according to data provenance. This early discovery of fault-inducing input records may help users revise their code before other pending (larger) traces. Line 6 in Algorithm 1 sorts the backwards-trace queue in an ascending order.

Overlapping Backward Traces. Multiple faulty output records may be caused by the same in-

put records due to operators such as `flatMap` or `join`, where a single data record can produce multiple intermediate records, leading to multiple faulty outputs. For example, in Figure 4.7, a fault-inducing input record `99504 , 01/01/1993 , 70in` generates more than one faulty output records, *i.e.,* `AK , 01/01 , 21251` (Figure 4.7(a)) and `AK , 1993 , 21251` (Figure 4.7(b)). While the cardinality of the individual backward trace from the faulty output is 4 and 3 respectively, the overlap of the two traces contains only two input records, leading to the two different faulty outputs (Figure 4.7(c)). The benefit of this prioritization is twofold. First, BIGSIFT prioritizes the common input records leading to multiple outputs before applying DD to records that are pertinent to fewer faulty outputs. Second, the intersection of two sets might help us to tighten the scope of DD application, avoiding redundant work for the same failure-inducing records.

To check the eligibility for this optimization, BIGSIFT explores the DAG of the Spark program to find at least one 1-to-many or many-to-many operator such as `flatMap` and `join`. The overlap is performed right after Phase II's "smallest job first", as shown by line 9 in Algorithm 1. BIGSIFT overlaps the two smallest backward traces (let's say t_1 and t_2) from the sorted queue, C_L, to find the intersection, $t_1 \cap t_2$ (line 45). If the test function evaluated over the execution of $t_1 \cap t_2$ 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 $t_2 - t_1$ (lines 47-48). 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 overlap/intersection trace twice. Conversely, this prioritization could waste time for computing the overlap when the two backward traces do not overlap, or when the overlap trace does not cause any faulty output.

4.3.3 Phase III: Optimized Delta Debugging

Based on the order prioritized by Phase II, BIGSIFT applies DD to each backward trace (lines 16-29 of Algorithm 1). BIGSIFT provides a universal splitting function, which allows DD to deterministically split an input configuration into n sub-configurations (line 17). Each sub-configuration is

then sequentially submitted for execution (line 18) until either a faulty sub-configuration is found (line 19), or all the sub-configurations pass the test (line 21). In the former case, DD is recursively called over the faulty sub-configuration. In the latter instead, each sub-configuration is used to compute a complement (lines 21-22) which are then executed and tested (line 23). If all the complements pass the test, DD either generates twice as many sub-configurations as before or n (size of original configuration) sub-configurations, which ever is smaller (line 27). It then starts testing these sub-configurations as explained earlier. Otherwise, if any one of the complement fails the test, DD starts exploring that sub-configuration (line 25).

Re-running a program on a large dataset can be extremely expensive. Next we describe two optimizations.

Bitmap Based Memoization of Test Results. In our running example from Figure 4.2(b), **Run 4** and **Run 7** test the same input configuration twice while applying delta debugging. DD is not capable of detecting redundant trials of the same input configuration and therefore tests the same input configuration multiple times. To avoid waste of computational resources, BIGSIFT uses a *test results memoization* optimization. A naive memoization strategy would require scanning of the content of an input configuration to check whether it was tested already; such configuration content-based memoization would be time consuming and not scalable. BIGSIFT instead leverages *bitmaps* to compactly encode the offsets in the original 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 (line 40 in Algorithm 1). 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 [90].

Adaptive Local Job Scheduling. When we investigate the debugging time spent for each run in DD and the number of input records, we discover that for a DD job small enough to be run on a single machine (*e.g.,* less than 5000 records), running it on a cluster is unnecessary. BIGSIFT schedules a DD run on either the cluster or on a local machine, as shown in lines 36-39 in Algorithm 1.

```
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 4.8: 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.

4.4 Tool Interfaces

In this section, we will walk through the tool features of BIGSIFT with focus on tool demonstration. 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 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 4.8 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:

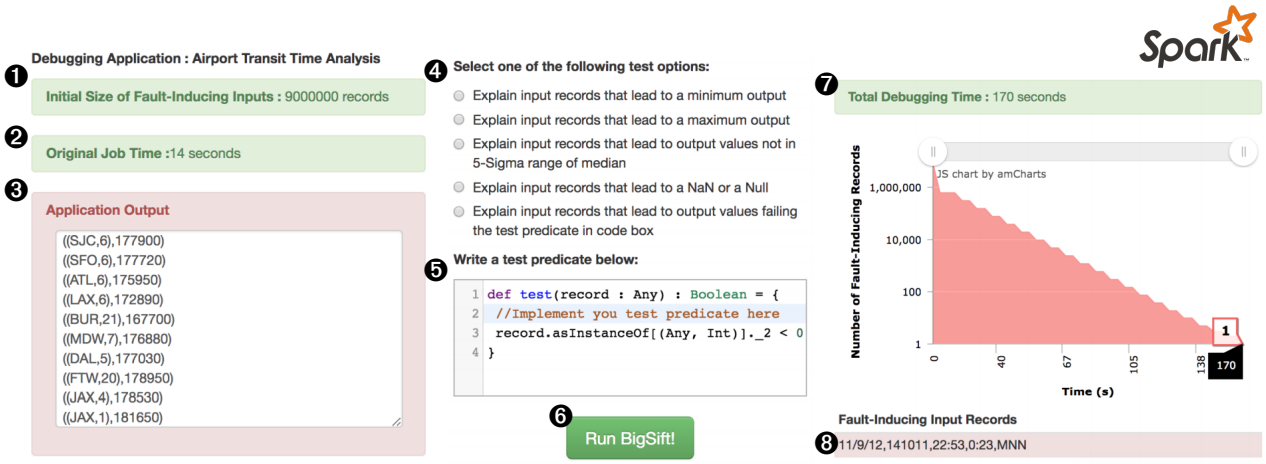


Figure 4.9: BIGSIFT's web-based interactive user interface

```
((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
```

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

Figure 4.10: BIGSIFT's API

Implementation To enable BIGSIFT, Alice can instantiate `BigSift` class with `SparkContext` and input file path as input arguments, as shown in Figure 4.10. Internally, this class instantiates `LineageContext` that enables Titian's instrumentation for data provenance support. 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. BIGSIFT's interactive UI is available on port 8989 on the Spark driver node. Figure 4.9 shows the web-based user interface.

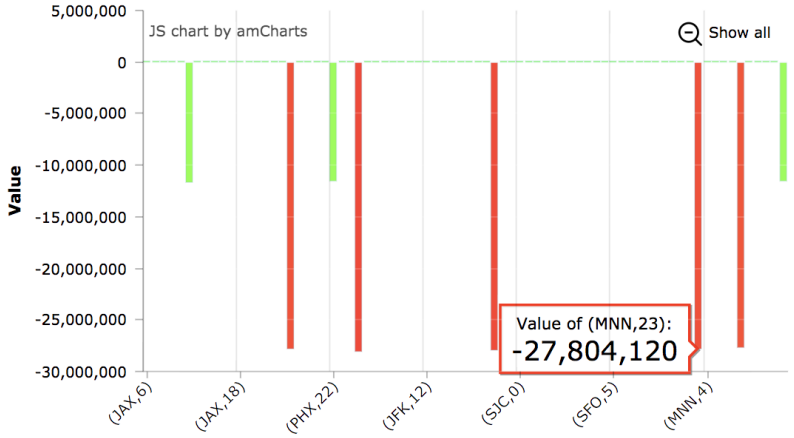


Figure 4.11: BIGSIFT's histogram visualization of key-value based output records

Step 1: Program Output Inspection Figure 4.9 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 4.9 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 4.11). For example, Alice can mark any negative value as incorrect using a histogram and note down this threshold to construct a test function.

noindent**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 4.9-❹) 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 4.9-❻). Internally, BIGSIFT selects the corresponding pre-defined test function to initiate debugging.

Step 3: Automated DISC Debugging

When BIGSIFT is invoked by the user, a realtime area chart appears on the UI. In Figure 4.9's chart on the right , 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 4.9-❼).

4.5 Evaluation

We perform a wide range of systematic experiments to evaluate BIGSIFT's runtime performance and precision of pinpointing fault-inducing input records compared against delta debugging and data provenance alone. To further differentiate the performance benefits from each optimization and prioritization, we design several versions of BIGSIFT as seen in Table 4.2: BIGSIFT-T simply combines delta debugging (DD) and test driven provenance (TP), BIGSIFT-O and BIGSIFT-S enable trace overlapping and smallest job first respectively in addition to leveraging both DD and TP. BIGSIFT-M applies bitmap based memoization of test results. Finally, BIGSIFT enables all optimization and prioritization heuristics. Our investigation addresses the following evaluation

#	Subject Programs	Source	Input Size	# of Ops	Program Description	Input Data Description	Fault Location
S1	Movie Histogram	PUMA	30 GB	4	Counts the number of movies in each rating category using `map`, `reduceByKey`, and `filter`	Movies with corresponding ratings from raters	Code
S2	Inverted Index	PUMA	40 GB	5	Generates a word-to-document indexing of a text data using `flatmap`, `map`, and `reduceByKey`	Text data with corresponding file id	Code
S3	Rating Histogram	PUMA	30 GB	4	Generates the frequency of each rating score from raters using `flatmap`, `map`, and `reduceByKey`	Movies with corresponding ratings from raters	Code
S4	Sequence Count	PUMA	80 GB	5	Counts the occurrence of every 3-word sequence using `flatmap`, `map`, and `reduceByKey`	Text data from Wikipedia dump	Code
W1	Rating Frequency	Custom	30 GB	4	Counts the number of ratings from each rater using `flatmap`, `map`, and `reduceByKey`	Movies with corresponding ratings from raters	Code
W2	College Student	Custom	4 GB	4	Finds the average age of all the students per college year using `map` and `groupbykey`	Student data with name, year, and date of birth	Data
W3	Weather Analysis	Custom	20 GB	4	Finds, in each state, the delta between the minimum and maximum snowfall reading for each day of any year and for any particular year using `flatmap`, `map`, and `groupbykey`	Daily snowfall measurements for every zipcode in feet and millimeters	Data
W4	Transit Analysis	Custom	20 GB	4	Finds the total layover time of all passengers spending less than 45 minutes per airport and per hour using `map`, `filter`, and `reduceByKey`	Passenger's arrival and departure time along with the airport code and date	Code

Table 4.1: Subject programs with input datasets

questions:

- How much improvement in the precision of fault-inducing input records does BIGSIFT provide in comparison to data provenance?
- How much improvement in the debugging time does BIGSIFT provide in comparison to delta debugging?
- When a time limit is set for fault localization, what are the benefits of *trace overlapping* and *smallest jobs first* prioritization heuristics respectively?

Evaluation Environment. We use a cluster consisting of sixteen i7-4770 machines, each running at 3.40GHz and equipped with 4 cores (2 hyper-threads per core), 32GB of RAM, and 1TB of disk capacity. The operating system is a 64bit Ubuntu 12.04. The datasets are all stored on HDFS version 1.0.4 with a replication factor of 3. The level of parallelism was set at two tasks per core. This configuration allows us to run up to 120 tasks simultaneously. BIGSIFT currently supports Apache Spark version 1.2.1 and leverages Titian to support data provenance in Spark. The runtime

overhead of lineage capture from Titian is reported to be below 30% [82].

Name	DD	TP	Trace Overlap	SJF	MEM
BIGSIFT-T	✓	✓	✗	✗	✗
BIGSIFT-O	✓	✓	✓	✗	✗
BIGSIFT-S	✓	✓	✗	✓	✗
BIGSIFT-M	✓	✓	✗	✗	✓
BIGSIFT	✓	✓	✓	✓	✓

Table 4.2: BIGSIFT with various optimizations. TP, SJF, and MEM stand for test driven provenance, smallest job first, and test results memoization, respectively.

Subject Programs. We evaluate BIGSIFT using a comprehensive set of subject programs and custom real-world workflows. We use eight subject programs in total, four of which are adapted from MapReduce PUMA benchmark [15]. PUMA benchmark provides an extensive set of big data processing applications along with a large-scale dataset for Hadoop MapReduce frameworks. We also developed four custom Spark programs (W1) Rating Frequency, (W2) College Student Analysis, (W3) Weather Analysis, and (W4) Air Transit Analysis. Table 4.1 shows all the subject programs along with their description. All subject programs except W2, W3, and W4 use the dataset provided by PUMA Benchmark. In W2, W3, and W4 we generate our own datasets using data generation scripts whereas W1 uses the PUMA dataset.

Test Functions. Each of the subject programs is also accompanied with a test function that checks for the correctness of each output record. This is analogous to writing an assertion or a unit test case in software engineering. Knowing the validity of each output record does not necessarily mean that a user can identify a minimum subset of failure-inducing input records. For most programs, the test function checks if individual output records are within valid ranges. For example, the test functions for S3 and S4 check that the count is positive for each rating and for 3-word sequences respectively. As another example, the test function for W2 checks that the average age of students of each college year is between 16 and 26.

Seeding Faults. The subject programs and their corresponding datasets used to evaluate BIGSIFT do not contain any faults. Therefore, we either seed faulty data records in the input dataset or inject programming errors in the subject program's code. These two types of faults in our experiments

underline the important distinction between data cleaning and debugging. Outliers or ill-formatted data records may be localized by intelligent data cleaning techniques; however, such data cleaning techniques cannot handle situations where the notion of faulty data keeps changing, depending on an application coding error. Given a test function, BIGSIFT not only finds inconsistently formatted records in the input data but also isolates cleanly formatted records interacting with faulty code, resulting in faulty outputs. The last column shows whether a fault is injected in data vs. code.

To inject faulty data, we select a random input record and modify it differently for each input dataset. For example, in the case of weather telemetry data, we randomly pick a single input record and replace the value of the snow measurement with the value in the unit of inches. This fault affects the final output of W3 and fails a check that the delta snowfall reading should not exceed 6000 millimeters. Similarly, in the college student data analysis W2, the date of birth for a randomly selected student is mutated to the date "0/0/0", which leads to a test failure.

We introduce code faults by modifying program logic—*i.e.,* code faults are introduced in the user-defined function of a data transformation operator such that the program behaves differently for certain intermediate data records. For example, in S4, the map transformation is modified, so that whenever two 3-sequence words "He has also" and "Romeo and Juliet" appear together in a line, the count of "He has also" is replaced with -99999. Similarly, in the subject program W4, an injected code fault affects a small set of intermediate records leading to a wrong value for the delta between the arrival and departure time of a passenger. For this case, the input data do not contain any data format anomaly or outliers. Six out of our eight subject programs contain code faults that cannot be debugged by data cleaning techniques because the notion of unclean data is dependent on coding faults.

4.5.1 Fault Localizability

To evaluate the ability to precisely localize fault-inducing input records, we measure the final size of the fault-inducing inputs from BIGSIFT. We also compare test-function driven data provenance (TP) with using data provenance (DP). The results are presented in Figure 4.12. The x-axis repre-

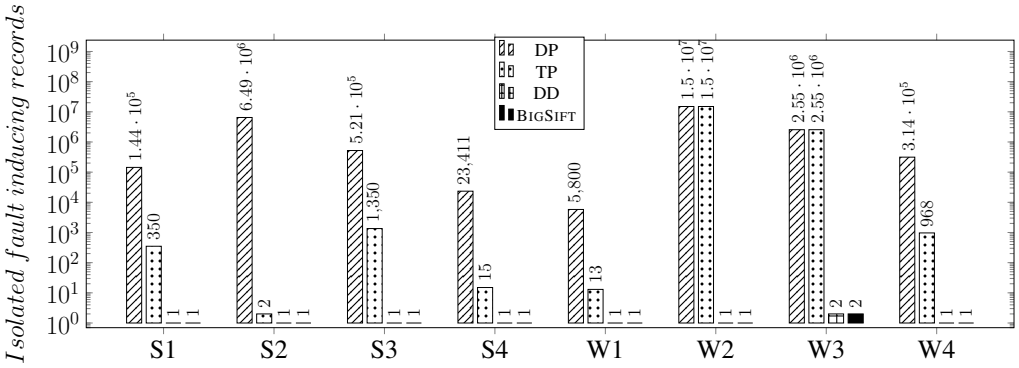


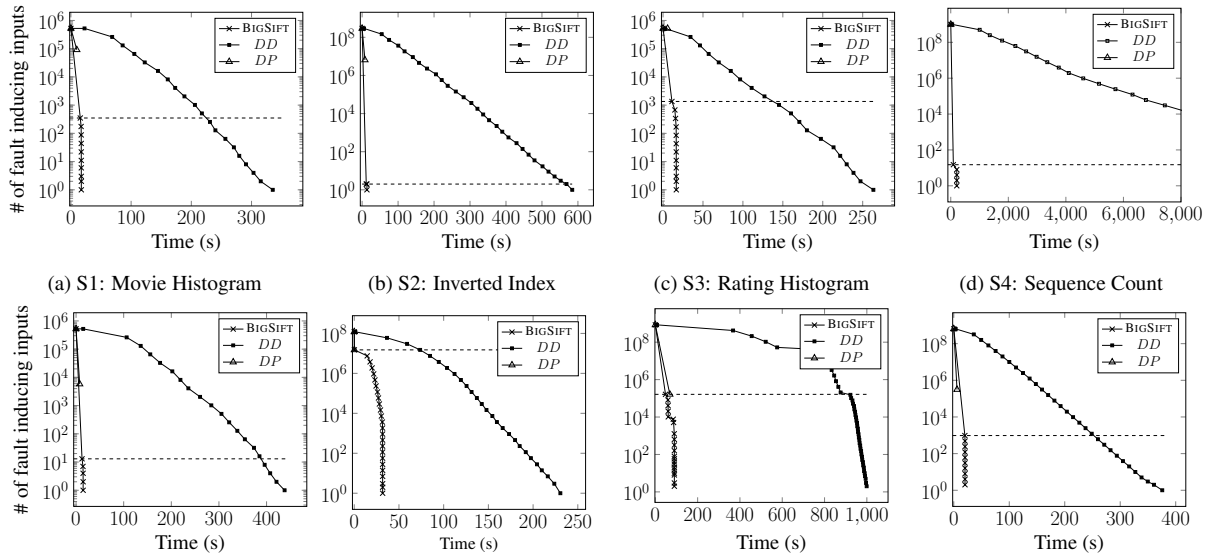
Figure 4.12: Fault localizability comparison on subject programs

sents the subject programs, while the y-axis measures the number of fault-inducing input records from BIGSIFT, DP, DD, and TP for each program. In almost all cases, data provenance over-approximates the fault-inducing input records, stopping at the order of 10^3 to 10^7 records, which is infeasible for programmers to manually sift through. For example, in program W2, DP is not able to localize fault-inducing input records beyond 15 million records. The poor localizability of DP is due to the use of `groupByKey` where the number of unique keys are only four possible keys, which results in over-approximating the scope of fault-inducing input records. On the other hand, we leverage test function push down in TP, when applicable, to reduce the size of fault-inducing input to a few thousand records (*e.g.,* in S1 and S3) by identifying faulty partitions (see dotted bars in Figure 4.12). BIGSIFT leverages DD to continue fault isolation after TP, achieving even higher accuracy.

4.5.2 Debugging Time

To evaluate the performance improvement of BIGSIFT, we compare the total debugging time of BIGSIFT against the baseline delta debugging (DD) and data provenance (DP). At every single iteration of DD, we log three metrics—the number of program runs (*i.e.,* iterations), the number of the fault-inducing input records, and the corresponding time span. These metrics help us analyze the runtime behavior at a fine-grained level. We then apply BIGSIFT on the same input data to localize the precise failure-inducing input records.

Figure 4.13 shows the performance improvement in BIGSIFT compared to original DD. The x-



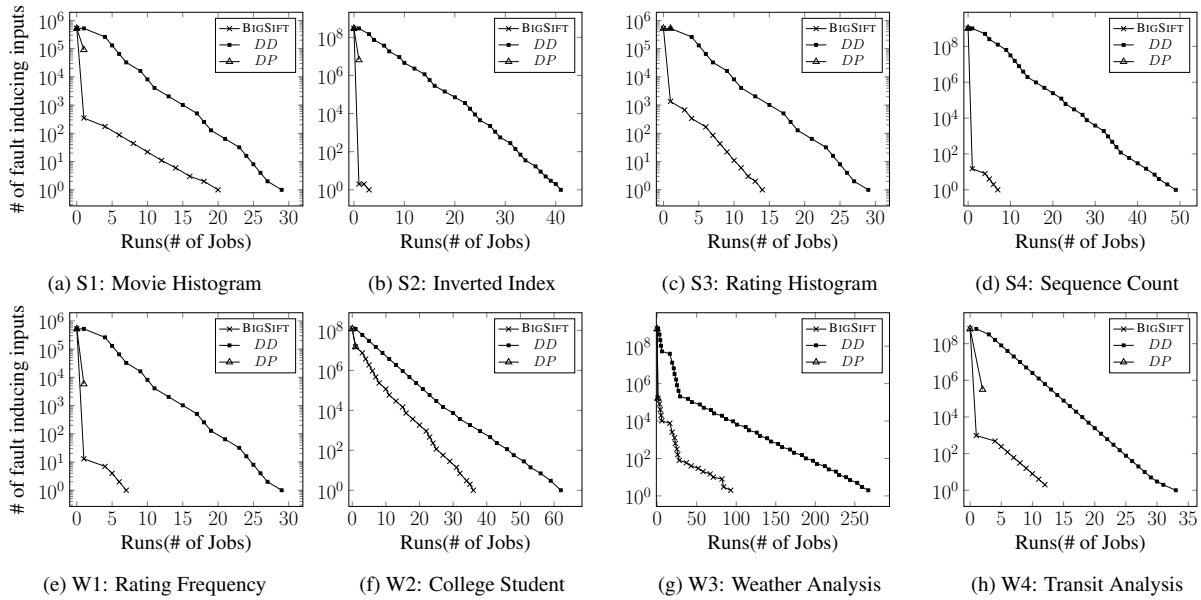
(a) S1: Movie Histogram (b) S2: Inverted Index (c) S3: Rating Histogram (d) S4: Sequence Count

(e) W1: Rating Frequency (f) W2: College Students (g) W3: Weather Analysis (h) W4: Transit Analysis

Figure 4.13: Performance comparison

axis represents the total debugging time in seconds and the y-axis represents the number of localized fault-inducing input records. For example, in Figure 4.13d, BIGSIFT takes 208 seconds to find the seeded fault, whereas DD takes 13772 seconds. DP stops after finding 23411 fault-inducing records in 398 seconds but cannot localize further from there. Comparison with DD shows that BIGSIFT enhances the debugging time by 66X. Further analysis shows that by applying test function driven data provenance (TP), BIGSIFT reduces the initial scope of fault-inducing records from more than 1 billion to just 15 records (dotted horizontal line) in 208 seconds, whereas DD takes 12395 seconds to achieve the same reduction. This significant decrease can be also seen in the other plots of Figure 4.13, as a steep drop till the dotted horizontal line, compared to the slow and steady elimination from DD marked in black. Figure 4.14 represents the number of runs required to perform fault localization. Figure 4.14d shows the result on S4. BIGSIFT takes just 7 runs to reach the minimum fault-inducing records, while DD takes 49 runs to achieve the same.

Table 4.3 shows the overall reduction in debugging time in BIGSIFT in comparison to DD. Overall, BIGSIFT provides from up to a 66X speed up in the total debugging time, in comparison to DD. In the case where TP does not significantly reduce the size of the initial fault-inducing input, the speed up is 7.4X. Interestingly, *the time taken for automated debugging of a singly faulty*



(a) S1: Movie Histogram (b) S2: Inverted Index (c) S3: Rating Histogram (d) S4: Sequence Count

(e) W1: Rating Frequency (f) W2: College Student (g) W3: Weather Analysis (h) W4: Transit Analysis

Figure 4.14: Reduction in the number of runs

Running Time (s)		Debugging Time (s)		
Program	Original Job	DD	BIGSIFT	Improvement
S1	56.2	232.8	17.3	13.5X
S2	107.7	584.2	13.4	43.6X
S3	40.3	263.4	16.6	15.9X
S4	356.0	13772.1	208.8	66.0X
W1	77.5	437.9	14.9	29.5X
W2	53.1	235.2	31.8	7.4X
W3	238.5	999.1	89.9	11.1X
W4	45.5	375.8	20.2	18.6X

Table 4.3: Fault localization time improvement

output in BIGSIFT *on average is 62% less than the time taken for a single run on the entire data* (Columns Original Job vs. BIGSIFT). With only up to 30% overhead incurred by Titian for lineage capture [82], BIGSIFT dramatically reduces the scope and cost of iterative fault localization, by leveraging the lineage mappings.

The reason behind this feasibility of automatic debugging is that, in many subject programs, BIGSIFT reduces the scope of fault-inducing input records by testing partially-aggregated results such that the later time spent on repetitive fault isolation in DD could be much smaller than the original time taken for the first run on the entire data. In fact, our result suggests that automated debugging can be brought to a reality more easily for data flow programs running in the DISC environments than other types of traditional C, C++, or Java applications, because debugging DISC

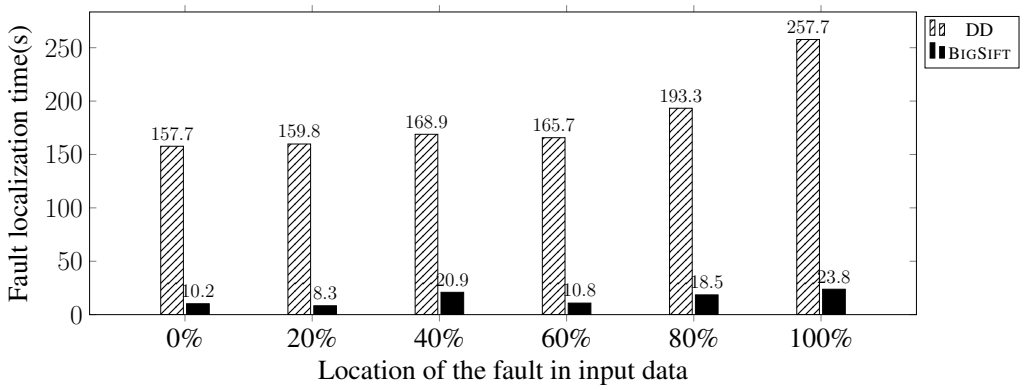


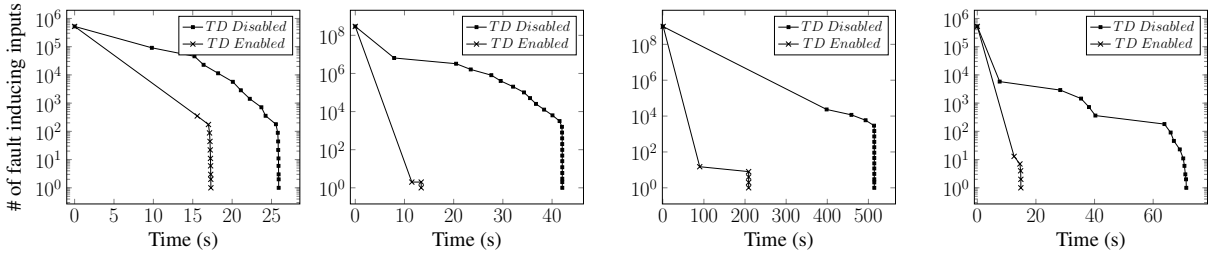
Figure 4.15: Fault localization time of BIGSIFT and DD for S1 w.r.t the location of seeded fault in input data.

workloads provide unique opportunities for systems-level optimizations.

Impact of Fault Location. While applying DD, the location of a faulty input record affects the total debugging time, because DD needs to test two sub-configurations sequentially at every iteration. If the first of the two always fails the test, DD will focus its search on the first one. Therefore, both DD and BIGSIFT may increase debugging time, if a fault-inducing record is located near the end of input data.

To evaluate the impact of fault-inducing input location on debugging time, we compare BIGSIFT with DD while varying the location of a fault-inducing input. Figure 4.15 summarizes the results where the x-axis represents the location of a fault (*e.g.,* 20% denotes that the fault is at one-fifth of the data) and the y-axis represents debugging time. When the location of fault-inducing input is changed from the start to the end (0% to 100%) with the increment of 20%, the debugging time of BIGSIFT increases from 10.2 seconds to 23.8 seconds for subject program S1. We also observe a similar trend in DD when the location of a fault is near the end of the input data.

Effects of Test Function Push Down. To evaluate the effects of test function push down (TD), we compare BIGSIFT with TD disabled vs. TD enabled. In the TD enabled version, BIGSIFT pushes down a user-defined test function to each individual partition to test partial results. Our evaluation targets subject programs whose last operator is `reduceByKey` (i.e., programs S1, S2, S3, S4, W1, and W4). For subject programs W2 and W3, the UDF of the last operator is not associative.



(a) S1: Movie Histogram (b) S2: Inverted Index (c) S4: Sequence Count (d) W1: Rating Frequency

Figure 4.16: Effect of test function push down (TD).

For example, in W2, the last transformation computes the average of each group. Computing an average is a non-associative operation; therefore, TP becomes basic data provenance.

Figure 4.16 illustrates how BIGSIFT completes fault localization faster than BIGSIFT without TD. In subject program S1 (Figure 4.16a), BIGSIFT takes 17 seconds to localize fault-inducing input records, 33% less than BIGSIFT with TD disabled. By testing partial results and applying DP afterwards on faulty partitions, BIGSIFT reduces the scope of fault-inducing input to just 350 records, while disabling TD reduces the scope to 91308 records.

4.5.3 Debugging Program Faults

As BIGSIFT is built on DD, by construction, it has the following characteristics:

- BIGSIFT does not enumerate all possible explanations. Instead, it finds a *single* minimum subset responsible for producing the same test failure. In other words, if there are two possible explanations of failure-inducing inputs, it finds one not both.
- BIGSIFT only guarantees to produce the same test failure when applying the given test function to the resulting set of fault-inducing input records. It may not produce the same faulty output value as the original failing run on the entire input.
- BIGSIFT is extremely beneficial for the case of finding *a needle in a haystack. i.e.,* both fault-inducing input and faulty output occur very rarely. Such debugging scenario is generally the most difficult case in software engineering, as developers cannot easily find a small, manageable size of data to reproduce the same failure symptom.

Program Versions	Original Job Time	Faulty Outputs	Data Affected	Debugging Time(s)				Fault Localization			
				BIGSIFT	DP	DD	BIGSIFT vs. Job Time	BIGSIFT	DP	DD	Improvement
W4-1	48.8	1020	555464920	12442.0	19.5	>12 hr	255X	1020	283790715	1020	$2.8x10^5$X
W4-2	46.6	367	37642315	2658.5	9.3	>12 hr	57X	367	56789568	367	$1.6x10^5$X
W4-3	45.5	170	33320879	1144.1	8.8	>12 hr	25X	170	43318865	170	$2.6x10^5$X
W4-4	46.5	1	1	8.5	8.4	431	0.18X	1	84948	1	$8.5x10^4$X

Table 4.4: Performance and fault localization of BIGSIFT on 4 versions of subject program W4 each with difference coding fault.

To manifest these strengths and limitations empirically, we design an experiment where we inject four different coding faults in subject program W4. These coding faults interact with a different amount of input records and produce different numbers of faulty output records in the final result. We compare performance and fault localizability with DD, DP, and the original running time.

```
  val diff = input.map{ data =>
      val arr_min = getMinutes(data._2)
      val dep_min = getMinutes(data._3)
      var timediff = dep_min - arr_min
      //Branch removed to inject code fault
-     if(timediff < 0 ){
-       timediff = 24*60 + timediff
-     }
      timediff
    }
```

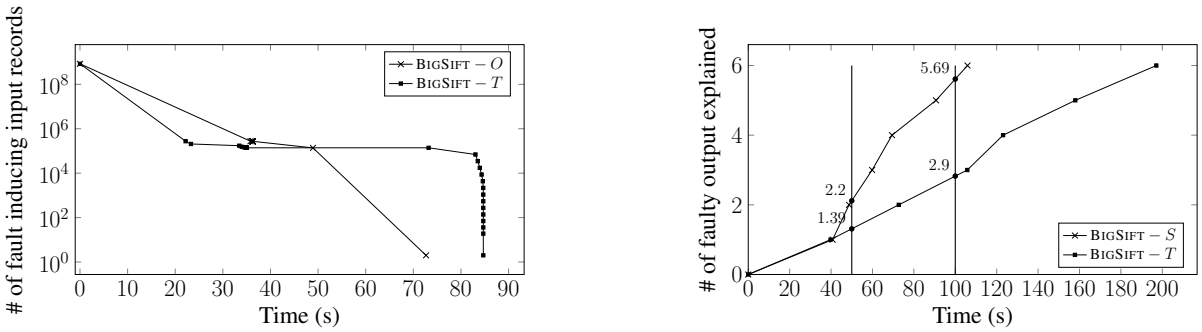
Figure 4.17: A branch is removed in subject program W4 to inject a code fault

The program W4 calculates the total transit time of all passengers who spend less than 45 minutes at each airport grouped by every hour. The input dataset used by the program is completely clean *i.e.,* the dataset is free from any kind of formatting anomalies or outliers. The four different versions of W4 are listed in Table 4.4. We count the number of faulty output records using differential testing by comparing the final results of the faulty version against the original program. Depending on code faults, the number of faulty outputs ranges from 1 to 1020 faulty outputs (Faulty Outputs). We conservatively estimate the number of faulty input records by profiling individual input records exercised by the faulty code region. This number varies from a single record to several million records (Data Affected). Figure 4.17 shows an example code fault from program W4-3 that removes a code fragment, re-adjusting the transit period over midnight. When there are multiple faulty output records, we run BIGSIFT and DD for each faulty output record in iteration. Table 4.4 summarizes the experiment results.

Consider the program version W4-1 that touches 555 millions input records and then generates 1020 faulty outputs. The entire process for debugging all 1020 faulty outputs takes 12442 seconds and the total time is 255X of the original job time. While the code fault touches 555 million input records, BIGSIFT finds only 1020 faulty inputs, each of which corresponds to reproducing the test failure of a single faulty output. It is because the goal of Delta Debugging is to find a minimum set of fault-inducing records that can reproduce each test failure, not to enumerate all possible explanations for each failure.

Nevertheless, BIGSIFT still performs better than DD which will take an estimated 4 days ($\geq$ 100 hours) to find the equal number of fault-inducing inputs. In our experiments, we use a cut-off time of 12 hours for DD. DP finds more fault-inducing inputs than BIGSIFT due to over-approximation, but the resulting set will also include non-faulty input data. For version W4-4, BIGSIFT finds one and only fault-inducing input record precisely in 8.5 seconds, which 82% less than the original job time. This is the kind of *a needle in a haystack* situation where existing techniques take a very long time to debug. DP takes 8 seconds but fails to localize the fault-inducing input after reaching 84K records. The result shows that BIGSIFT performs well in terms of localizing fault-inducing input and reducing debugging time, when both the affected inputs and faulty outputs are highly infrequent, which is often the most challenging case of debugging.

In practice, a developer is unlikely to use BIGSIFT to provide all failure-inducing input records for all individual faulty outputs simultaneouly. Normally, a developer starts a debugging task with investigating the root cause of one faulty output and then fixes the source of the error before moving onto investigating the next faulty output. Therefore, in practice, when the program version W4-1 produces 1020 faulty outputs, we do not expect that a developer will run BIGSIFT for 1020 iterations to find failure-inducing input records for individual faulty outputs all at once. In fact, several faulty output records (or several test failures) are often caused by a single code fault or similar data faults. Therefore, a fix for a single fault may remove more than one faulty output. The results summarized above illustrate a very conservative and exceptional debugging scenario, where a developer wants to find the source of all individual faulty outputs at once without fixing



(a) Trace Overlapping

(b) Smallest Job First

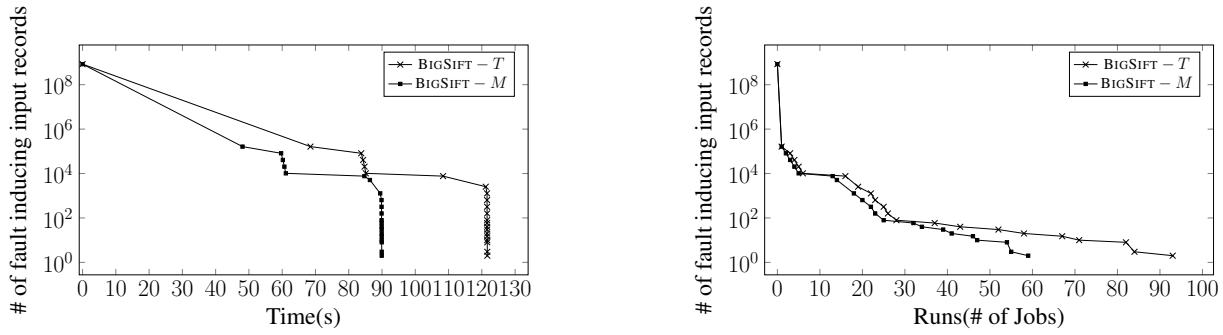
Figure 4.18: Benefits from Trace Overlapping (a) and Smallest Job First (b)

the discovered errors along the way.

4.5.4 Optimization and Prioritization Effect

To assess the benefits from each optimization and prioritization heuristic, we design four different versions of BIGSIFT as illustrated in Table 4.2. Each variation was evaluated on all subject programs, unless not applicable.

Trace Overlapping. We evaluate *trace overlapping* in BIGSIFT-O to assess its prioritization benefit, when there are multiple faulty output records. The benefits of overlapping the traces is debugging time reduction by prioritizing the common failure-inducing inputs that may be responsible for multiple faulty outputs. To assess whether this prioritization achieves any time saving, we compare BIGSIFT-O vs. BIGSIFT-T on subject programs W3, where we have 2 faulty output records. This program includes a `flatMap` operator which propagates a single faulty record into multiple faulty records. We observe (1) the number of fault-inducing input records identified within the same time limit and (2) the overall improvement in debugging time. In Figure 4.18a, BIGSIFT-O produces the exact same set of fault-inducing input records in 86% of the time, compared to BIGSIFT-T, by saving the time to identify the 1158 overlapping failure-inducing records twice. Figure 4.18a shows that BIGSIFT-O incurs an initial cost of computing the intersection. However, the remaining of the two overlapped traces do not contain the fault, which saves the fault localization time by not applying DD on them. The benefit of this prioritization is notable especially TD is not applicable.



(a) Fault Localization Time

(b) Number of Runs

Figure 4.19: Benefits of Bitmap Based Memoization w.r.t fault localization time (a) and number of DD runs (b)

Smallest Jobs First. BIGSIFT-S prioritizes backward traces in a *smallest job first* manner in an ascending order of the cardinality of backward traces from data provenance. This prioritization improves the coverage of faulty outputs when there are multiple faulty outputs to explain within the same time limit. We compare the coverage of the faulty outputs with BIGSIFT-S and BIGSIFT-T on program W3 with 6 faulty output records, where BIGSIFT-T selects traces at random.

Figure 4.18b illustrates the comparison. The y-axis represents the number of faulty output records explained, and the x-axis represents the time spent to perform these tasks. The reference lines at 50 and 100 seconds represent different possible time limits. By prioritizing DD on the cardinality of the scope of potential failure-inducing input records, BIGSIFT-S explains 5 faulty output records in W3, whereas the baseline BIGSIFT-T explains only 2 faulty output records with 100 seconds as the time limit.

Bitmap Based Memoization of Test Results. When applying DD in Phase III, in order not to test the same input sub-configuration multiple times, BIGSIFT-M uses the *test results memoization* optimization by maintaining a list of configuration descriptions (bitmaps) and the corresponding test outcomes.

To evaluate the advantage of this optimization, we compare BIGSIFT-M with BIGSIFT-T on program W3. Figure 4.19(b) shows the comparison in terms of the number of DD runs where the x-axis represents the number of jobs executed and the y-axis represents the size of the fault-inducing

input set. BIGSIFT-M eliminates 34 duplicate tests in Phase III by caching test results. BIGSIFT-M needs 59 runs to find the minimum fault-inducing input, whereas BIGSIFT-T needs 93 runs to get the same result. The savings with respect to DD runs is also reflected as reduction in the debugging time of BIGSIFT-M. Figure 4.19(a) shows that BIGSIFT-M takes 89 seconds as opposed to 121 seconds for BIGSIFT-T to localize the minimum fault-inducing input. On program W3, test memoization reduces the debugging time by 26%.

4.6 Discussion

This chapter describes BIGSIFT that combines insights from both data provenance in the database and fault-localization in software engineering to bring automated debugging techniques closer to a reality for big data analytics. Our experiment results validate sub-hypothesis (**SH 2**). They show that automated debugging can be done in a *scalable* and *precise* manner by leveraging the semantics of data flow operators, the properties of data partitioning, and test-driven data provenance to reduce the scope of failure-inducing records before initiating delta debugging. While debugging help us belatedly identify failure's root cause, the developers of big data applications must catch such issues in the testing phase preemptively. Current testing practices for big data applications are either time-consuming (*e.g.,* testing on the entire input data) or ineffective (*e.g.,* test on a small sample data). This motivates us to explore ideas that will help developers detect bugs in their big data applications using the least amount of test data. In the next chapter, we investigate our sub-hypothesis (**SH 3**) and propose a new white-box test generation approach that makes local testing efficient and effective for big data analytics.

CHAPTER 5

White-box Testing For Big Data Analytics

The previous two chapters present algorithms that isolate the root cause of failures or incorrect output through interactive and automated debugging for big data analytics. However, there is a huge cost of such issues arising in the production both in terms of time and resources. Therefore, big data applications must be of the highest quality before they run on a cloud environment to reduce such occurrences. This chapter examines the following sub-hypotheses **SH3**: *By abstracting the implementation of dataflow operators and by modeling the semantics of user-defined functions in tandem, we can generate a small set of test inputs that are capable of revealing more defects than the entire input dataset.* To investigate this hypothesis, we propose a symbolic execution based white-box test generation technique for big data applications that abstracts the DISC framework's code into logical specifications of dataflow operator and combines it with the semantics of user-defined functions, while achieving high test quality and efficiency.

5.1 Introduction

Big data applications process large amounts of data and can run for days. Thus, it is expensive and time consuming to test them in the production setting. The standard practice for testing these applications remains running them locally on randomly sampled inputs, which is unlikely to yield test coverage. As a result, it is not uncommon to see big data applications fail in production due to untested corner cases, wasting the excessive amount of resources consumed by these applications before failing.

We present a systematic input generation technique, called BIGTEST, that embodies a new white-

```
1 val x,y,z;
2 if(x<y)
3   z = y/x; //PC1: x < y = true, Effect: z=y/x
4 else
5   z = x/y; //PC2: x >= y = true, Effect: z=x/y
```

Figure 5.1: Symbolic PathFinder produces a set of path constraints and their corresponding effects

box testing technique for big data applications. BIGTEST reasons about the *combined* behavior of UDFs with relational and dataflow operations. Instead of symbolically executing the implementation of dataflow operators, BIGTEST includes a logical abstraction for dataflow and relational operators when symbolically executing UDFs in the big data application. The resulting set of combined path constraints are transformed into SMT (satisfiability modulo theories) queries and solved by leveraging an off-the-shelf theorem prover, Z3 [51] or CVC4 [26], to produce a set of concrete input records.

We evaluate BIGTEST on 31 real-world faulty big data applications and show that BIGTEST can minimize data size for local testing by 10^5 to 10^8 orders of magnitude, achieving the CPU time savings of 194X on average, compared to testing code on the entire production data. It can reveal 2X more manually injected faults than prior approaches on average. With these results, we demonstrate that *interactive* local testing of big data analytics is feasible and that developers do not need to test their application on the entire production data; hence, supporting our sub-hypothesis (**SH 3**).

The rest of this chapter is organized as follows. Section 5.2 provides a brief introduction to symbolic execution. Section 5.3 describes a motivating example. Section 5.4 describes the design of BIGTEST. Section 5.5 demonstrates BIGTEST on a big data application. Section 5.6 describes evaluation settings and results. Section 5.7 concludes this chapter.

5.2 Background: Symbolic Execution

BIGTEST builds on Symbolic Java PathFinder (SPF) [114]. Internally, SPF relies on the analysis engine of Java PathFinder (JPF) model checking [135]. It interprets Java bytecode on symbolic

```
1  val trips = sc.textFile("trips_table.csv")
2    .map{  s =>
3  ❶     val cols = s.split(",")
4       (cols(1),cols(3).toInt/cols(4).toInt) }
5       //Returns location and speed
6  val zip = sc.textFile("zipcode_table.csv")
7    .map{  s =>
8  ❷     val cols = s.split(",")
9       (cols(1),cols(0)) }
10      // Returns location and its name
11   .filter{
12 ❸    s => s._ 2 == "Palms" }
13 val joined = trips.join(zip)
14 joined
15    .map{  s =>
16 ❹  if (s._2._1 > 40) ("car",1)
17       else if (s._2._1 > 15) ("bus",1)
18       else ("walk",1)
19   }
20   .reduceByKey(_+_) ❺
21   .saveAsTextFile("hdfs://...")
```

Figure 5.2: Alice's program estimates the total number of trips originated from "Palms."

inputs and produces a set of symbolic constraints. Each constraint represents a unique path in the program, and can be ingested by a theorem solver to generate test inputs. Figure 5.1 illustrates an example symbolic execution result. By attaching listeners to SPF, the path conditions and the effects of each path can be captured. For this program, SPF produces two path conditions: (1) the first path produces the effect of `z=y/x`, when the path condition `x<y` holds true and (2) the second path produces `z=x/y` as an effect, when the path condition $x \geq y$ is satisfied.

5.3 Motivating Example

This section presents a running example to motivate BIGTEST. Suppose that Alice writes a big data application in Spark to analyze the Los Angeles commuting dataset. She wants to find the total number of trips originating from the "Palms" neighborhood using: (1) a public transport whose speed is assumed to be faster than 15 but slower than 40 mph, (2) a personal vehicle which is estimated to be faster than 40 mph, and (3) on foot which is estimated as slower than 15 mph. Each row in the Trips dataset represents a unique identifier for the trip, the start and end location in terms of a zip code, the trip distance in miles, and the trip duration in hours, for example,

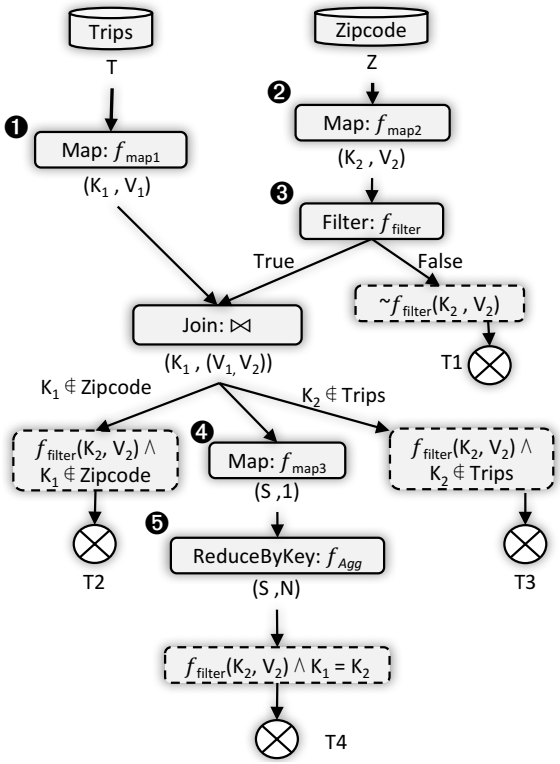
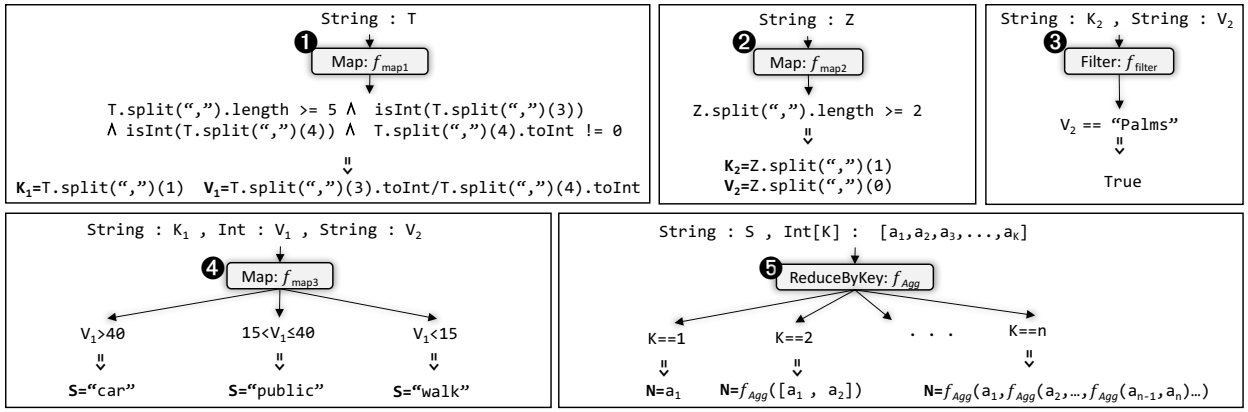


Figure 5.3: Dataflow operators' paths by BIGTEST. Solid and dotted boxes represent transformations and path constraints, respectively.

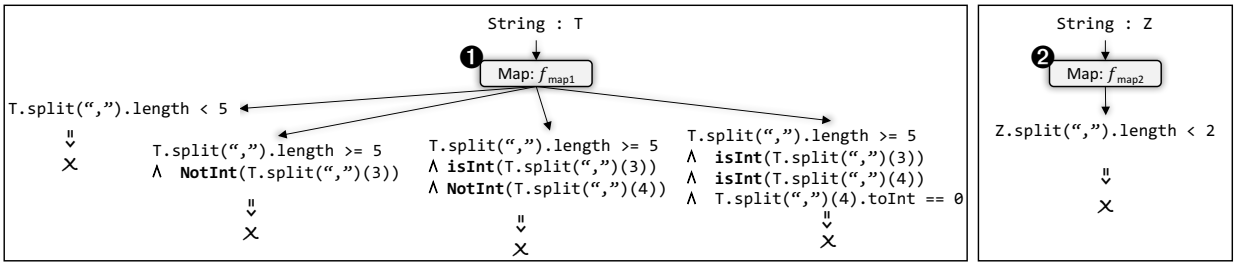
`1,90034,90024, 10, 1`. To map an area zip code to its corresponding area name, Alice uses another dataset that assigns a name to each zip code in the following manner: `90034,Culver City`

To perform this analysis, Alice writes a Spark application in Figure 5.2. She loads both datasets (lines 1 and 6), parses each dataset, selects the start location of a trip as a key, and computes the average speed as a value by dividing the distance by duration (lines 2-4). Alice outputs a zip code as a key and an area name as a value (lines 7-9) and filters the area name with "Palms" at line 12. She joins the two data sets (line 13). In the subsequent `map` operation (line 15-18), she categorizes the trips based on the average speed into three categories. She finally counts the frequency of each trip kind and stores them (lines 20 and 21). Though this program is only 21 lines long, it poses several challenges for modeling test paths.

Equivalence Classes of Dataflow Operators. Consider `filter` ❸ at line 11. To exhaustively test this operator, we must consider two equivalence classes: the first where a data record satisfies



(a) Non-terminating path conditions of individual UDFs



(b) Path constraints for terminating paths in UDFs

Figure 5.4: BIGTEST identifies path constraints for both non-terminating and terminating program paths while symbolically executing the program.

the filter and moves onto the next operator and the second where the filter does not satisfy and its data flow terminates. If we only model non-terminating case then test data would contain passing data records only and hence, would not detect a fault in which `filter` is removed from the big data application. To model `join` at line 13, we must have three equivalence classes—two terminating cases and one non-terminating case: (1) an input record in the left table (“Trip”) does not have a matching key on the right table (“ZipCode”), terminating its data flow, (2) an input in the right table does not have a matching key on the left, terminating its data flow, and (3) there exists a key that appears in both tables, passing the joined result to the next operator. Modeling such terminating cases is crucial otherwise test data generated produce the same output for both `join` and `leftOuterJoin` and do not reveal faults that are based on incorrect `join` type usage.

UDF Paths. Consider the `map` ❹ at lines 15-18. There are three internal path conditions: (1) speed $>$ 40 mph, (2) 15 mph $<$ speed $\leq$ 40 mph, and (3) speed $\leq$ 15 mph. The sub figure ❹ in

#	Constraint	Trips	Zipcode
C1	`T.split(",").length < 5`	`""`	`_ _`
C2	`T.split(",").length ≥ 5 ∧ NotInt(T.split(",")(3))`	`_, _, _, "", _`	`_, _`
C3	`T.split(",").length ≥ 5 ∧ isInt(T.split(",")(3)) ∧ NotInt(T.split(",")(4))`	`_, _, _, "-2", ""`	`_, _`
C4	`T.split(",").length ≥ 5 ∧ isInt(T.split(",")(3)) ∧ isInt(T.split(",")(4)) ∧ T.split(",")(4).toInt = 0`	`_, _, _, "-2", "0"`	`_, _`
C5	`Z.split(",").length < 2`	`_`	`""`
C6	`Z.split(",").length ≥ 2 ∧ V2 != "Palms"`	`_`	`_, "\x00"`
C7	`T.split(",").length ≥ 5 ∧ isInt(T.split(",")(3)) ∧ isInt(T.split(",")(4)) ∧ T.split(",")(4).toInt != 0 ∧ Z.split(",").length ≥ 2 ∧ V2 = "Palms" ∧ K1 ∉ Zipcode`	`_, "!0!", _, _, _`	`"\x00", "Palms"`
C8	`... ∧ V2 = "Palms" ∧ K2 ∉ Trips`	`_, ""!0!", _, _, _`	`"\x00", "Palms"`
C9	`... ∧ V2 = "Palms" ∧ K1= K2 ∧ V1 > 40`	`_, "\x00", _, "41", "1"`	`"\x00", "Palms"`
C10	`... ∧ V2 = "Palms" ∧ K1= K2 ∧ 15< V1 < 40`	`_, "\x00", _, "16", "1"`	`"\x00", "Palms"`
C11	`... ∧ V2 = "Palms" ∧ K1= K2 ∧ V1 < 15`	`_, "\x00", _, "0", "1"`	`"\x00", "Palms"`

Table 5.1: Generated input data where each row represents a unique path. Variables `T`, `Z`, `V`, and `K` are defined in Figure 5.3.

Figure 5.4a shows corresponding path conditions and effects.

String Constraints. To analyze the second `map` ❷ at lines 7 to 9, we must reason about the entailed string constraints. Given a string `Z` in sub figure ❷ in Figure 5.4a and 5.3, to split the data into two columns, it must satisfy a string constraint `Z.split(",").length` $\geq$ `2` to produce the effect where the key K_2 is `(Z.split(",") (1)` and the value V_2 is `(Z.split(",")(0))`. String manipulation is critical to many big data applications. In the above example, at least one test must contain a string `z` without delimiter `","`, so that `z.split(",")(1)` leads to ArrayIndexOutOfBoundsException which will then expose the inability of the UDF to handle exceptions. Otherwise, this application may crash in production, when the input record does not have an expected delimiter.

Arrays. To analyze `reduceByKey` ❺ at line 20 (also in Figure 5.4a), we must model how the UDF operates on the input array of size `K`, $[a_1, a_2, \ldots, a_K]$ and produces the corresponding output $f_{agg}(a_1 \ , \ f_{agg}(a_2 \ \ldots f_{agg}(a_{K-1}, a_K)\ldots))$. For example, the UDF `(_+_)` returns the sum of two input arguments. When the array size `K` is given by a user, the final output N is $a_1+(a_2+\ldots(a_{K-1}+a_K))$.

Summary. Due to the internal path conditions entailed by individual UDFs, instead of four high-level dataflow paths shown in Figure 5.3, Alice must consider eleven paths in total, which are enumerated in Table 5.1. Figure 5.4 shows the symbolic execution tree at the level of dataflow operators on the left and the internal symbolic execution trees for individual UDFs on the right.

Lastly, example data generated by BIGTEST for each JDU path using Z3 is shown in Table 5.1. While these example data records may not look realistic, such data is necessary to exercise the downstream UDFs that are otherwise unreachable with the original dataset. For instance, filtering a dataset without any passing data record will result in an empty set and consequently, the UDFs after the `filter` will never get tested with the original data. Therefore, synthetic data is necessary and crucial to expose downstream program behavior.

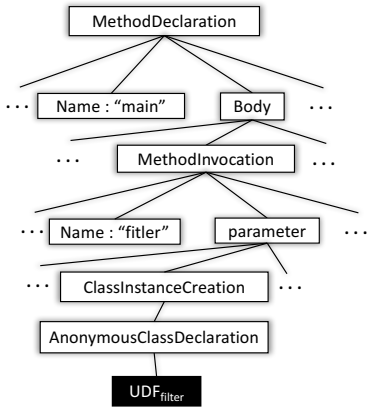
5.4 Approach

BIGTEST takes in an Apache Spark application in Scala as an input and generates test inputs to cover all paths of the program up to a given bound by leveraging theorem provers Z3 [51] and CVC4 [26].

5.4.1 Big Data Application Decomposition

A big data application is comprised of a direct acyclic graph where each node represents a dataflow operator such as `reduce` and corresponding UDFs. As the implementation of dataflow operators in Apache Spark spans several hundred thousand lines of code, it is not feasible to perform symbolic execution of a big data application along with the Spark framework code. Instead, we abstract the internal implementation of a dataflow operator in terms of logical specifications. We decompose a big data application into a dataflow graph where a node calls each UDF and combine the symbolic execution of the UDFs using the logical specification of dataflow operators.

UDF Extraction. BIGTEST compiles the big data application into Java bytecode and traverses each Abstract Syntax Tree (AST) to search for a method invocation corresponding to each dataflow operator. The input parameters of such method invocation are UDFs represented as anonymous functions as illustrated in Figure 5.5a. BIGTEST stores the UDF as a separate Java class shown in Figure 5.5c and generates a configuration file required by JPF for symbolic execution. BIGTEST also performs dependency analysis to include external classes and methods referenced in the UDF.



(a) Generated AST

```
sc.textFile("zipcode.csv").map{...}
  .filter{_._2 == "Palms"}
```

(b) Big Data Application

```
class filter{
 static void main(String args[]){
  apply(null);
 }
 static boolean apply(Tuple2 s){
   return s._2().equals("Palms")
 }
}
```

(c) Extracted Filter UDF

Figure 5.5: BIGTEST extracts UDFs corresponding to dataflow operators through AST traversal.

```
def f(a:Int,b:Int){
  return a+b;
}
//Usage in reduce
...reduce{f}
```

(a)

```
def f_reduce(arr:Array[Int]){
 var sum = 0;
 for(a <- 1 to K)//K is bound
   sum = udf(sum,arr(a));
 return sum; }
```

(b)

Figure 5.6: (a) a normal invocation of `reduce` with a corresponding UDF. (b) an equivalent iterative version with a bound K

Handling Aggregator Logic. For aggregation operators, the attached UDF must be transformed. For example, the UDF for `reduce` is an associative binary function, which performs incremental aggregation over a collection shown in Figure 5.6a. We translate it into an iterative version with a loop shown in Figure 5.6b. To bound the search space of constraints, we bound the number of iterations to a user provided bound `K` (default is 2).

5.4.2 Logical Specifications of Dataflow Operators

This section describes the equivalence classes generated by each dataflow operator's semantics. We use C_I to represent a set of path constraints on the input data, I, for a particular operator. A single element c in C_I contains path constraints that must be satisfied to exercise a corresponding unique path. We define f as the set of symbolic path constraints of a UDF where $f(t)$ represents constraints of a unique path exercised by input t. By abstracting the implementation of dataflow

OPERATOR	INPUTS	LOGICAL SPECIFICATION	
filter(udf)	I: Input Table $udf : t \rightarrow Bool$	Non-Terminating	❶ $\exists t$: $t \in I \wedge c \in C_I \wedge c(t) \wedge f(t)$
		Terminating	❷ $\exists t$: $t \in I \wedge c \in C_I \wedge c(t) \wedge \neg f(t)$
map(udf)	I : Input Table, O : Output Table $udf : t \rightarrow t'$ where $t' \in O$	Non-Terminating	❸ $\exists t$: $t \in I \wedge c \in C_I \wedge c(t) \wedge f(t)$
flatmap(udf)	I : Input Table, O: Output Table $udf : t \rightarrow Collection\ of\ t'$ where $t' \in O$	Non-Terminating	❹ $\exists t$: $t_I \in I \wedge c \in C_I \wedge c(t) \wedge f(t)$
join	R : Right Table, $t_R \in R$ L : Left Table, $t_L \in L$	Non-Terminating	❺ $\exists t_R, t_L$: $c_R \in C_R \wedge c_L \in C_L \wedge c_R(t_R) \wedge t_{R,key} = t_{L,key} \wedge c_L(t_L)$
		Terminating	❻ $\exists key, t_R$: $c_R \in C_R \wedge c_L \in C_L \wedge c_R(t_R) \wedge t_{R,key} = key \wedge (\forall t_L \in L : c_L(t_L) \wedge t_{L,key} \neq key)$
		Terminating	❼ $\exists key, t_L$: $c_R \in C_R \wedge c_L \in C_L \wedge c_L(t_L) \wedge t_{L,key} = key \wedge (\forall t_R \in R : c_R(t_R) \wedge t_{R,key} \neq key)$
groupByKey	I : Input Table $t \in I$ and $t = (t_{key}, t_{value})$	Non-Terminating	❽ $\exists t$: $t \in I \wedge c \in C_I \wedge c(t) \wedge \lvert\{X \lvert x \in I \wedge x_{key} = t_{key}\}\rvert > 0$
reduce(udf) reduceByKey(udf)	I : Input Table, O: Output $udf : (t, t) \rightarrow t'$ where $t' \in O$	Non-Terminating	udf' is an iterative version of the original UDF udf, given as an input to reduce/reduceByKey. f' represents the set of path constraints generated from symbolic execution of udf'. ❾ $\exists t_1, t_2, t_3, \ldots, t_n \in I : c_1, c_2, \ldots, c_n \in C_I \wedge c_1(t_1) \wedge c_2(t_2) \wedge \ldots.. \wedge c_n(t_n) \wedge f'(I)$

Table 5.2: C_I represents a set of incoming constraints from the input table I, where each constraint $c \in C_I$ represents a non-terminating path. $c(t)$ represents that record $t \in I$ must satisfy constraint c. f defines the set of path constraints generated by symbolically executing udf and $f(t)$ represents the path constraint of a unique path exercised by input tuple t.

operators into logical specifications, BIGTEST does not have to symbolically execute the Spark framework code (about 700KLOC), as it focuses on application level faults only as opposed to framework implementation faults which is out of the scope of this dissertation. BIGTEST supports all popular dataflow operators with the exception of deprecated operators such as `co-join`.

Filter. `Filter` takes a boolean function udf deciding if an input record should be passed to downstream operators or not. Therefore, we model two equivalence classes: (1) there exists a record t that satisfies udf and one of the incoming constraints C_I from input table I (*i.e.,* the table produced by its upstream satisfying operator), shown in ❶ Table 5.2; (2) there exists a record t that satisfies one of the incoming constraints but not udf, shown in ❷ Table 5.2.

Map and Flatmap. `Map` takes a UDF udf as an input and applies it to each input record to produce an output record. It has one equivalence class, where there exists tuple t from the input table I satisfying one of the incoming constraints, $c \in C_I$ and also one of the path constraints in f *i.e.,* path constraints generated by symbolically executing udf, shown in ❸ Table 5.2. `Map` is supported by the previous work SEDGE but SEDGE considers the UDF udf as a black box, uninterpreted function. `Flatmap` splits an input record using a udf to generate a set of records, and thus the equivalence class of `flatmap` is similar to that of `map`, as shown in ❹. BIGTEST handles `flatmap` by explicitly modeling a collection, described in Section 5.4.3.

Join. `Join` performs an *inner-join* of two tables t_r on the right and table t_l on the left based on

the equality of keys, assuming that records from both tables are of the type `Tuple` (key, value). We model the output records of `join` into three equivalence classes: (1) the key of tuple t_R in the right table matches with a key of tuple t_L on the left; (2) the key of tuple t_R in the right table does not match with any key of tuple t_L on the left; and (3) the key of tuple t_L in the left table does not match with any key of tuple t_R on the right. ❺, ❻, and ❼ in Table 5.2 represent the three equivalence classes.

Reduce and ReduceByKey. `reduce` takes a udf and a collection as inputs and outputs an aggregated value, while `reduceByKey` performs a similar operation per key. As discussed in Section 5.4.1, BIGTEST generates an equivalent iterative version of the udf with a loop. By this refactoring of udf to udf', the equivalence classes could be modeled similar to that of `map`, where there exist input records $t_1, t_2, \ldots, t_n \in I$ on which each of the corresponding non-terminating constraint $(c_1, c_2, \ldots, c_n) \in C_I$ from the input table I holds true. In addition, each record must satisfy the constraints of udf', satisfying $f'([t_1, t_2, \ldots, t_n])$, as shown in ❾ of Table 5.2.

5.4.3 Path Constraint Generation

This section describes several enhancements in Symbolic Path Finder (SPF) to tailor symbolic execution for big data applications. Big data 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.

```
1  def parse(s:String){
2   val cols = s.split(",")
3   (cols(0) , cols(1)) }
```

Figure 5.7: A UDF with string manipulation

Strings. Operations such as `split` for converting an input record into a key-value pair are common in big data applications but are not supported by SPF. BIGTEST extends SPF by capturing calls to `split`, recording the delimiter, and returning an array of symbolic strings. When an n-th element of this symbolic array is requested, SPF returns a symbolic string encoded as `splitn`

with a corresponding index. By representing the effect of Figure 5.7 as `(splitn(",",s,0), splitn(",",s,1))`, BIGTEST generates one terminating constraint, where s can only split into fewer than two segments, and one non-terminating constraint where s can split into at least two segments. Due to no `split` support, naïve SPF generates a string without any delimiter as a test input *e.g.,* "\x00" instead of "\x00,\x00". This input would lead to ArrayIndexOutOfBoundsException while accessing a string using `split(",")(1)`.

```
1 def agg(arr:Array[Int]){
2  val sum = arr(0); // Bound K=3
3  for(a <- 1 to min(arr.size,2)) sum += arr(a)<0 ? 0 : arr(a);
4  sum }
```

Figure 5.8: An iterative version of aggregator UDF

Collections. Constructing and processing collections through operators such as `flatmap` are essential in big data applications. Therefore, BIGTEST explicitly models the effect of applying a UDF on a collection. In Figure 5.8, an iterative version of aggregator logic produced by BIGTEST takes a collection as input and sums up each element, if the element is greater than or equal to zero. Given a user-provided bound `K=3` BIGTEST unrolls the loop three time and generates four pairs of a path condition (P) and the corresponding effect (E):

1. `P:a(1)<0` $\wedge$ `a(2)<0`, `E:a(0)`
2. `P:a(1)`$\geq$`0` $\wedge$ `a(2)<0`, `E:a(0)+a(1)`
3. `P:a(1)<0` $\wedge$ `a(2)`$\geq$`0`, `E:a(0)+a(2)`
4. `P:a(1)`$\geq$`0` $\wedge$ `a(2)`$\geq$`0`, `E:a(0)+a(1)+a(2)`

A naïve SPF does not handle collections well and thus may generate an array of length 1 only, not exercising line 3 in Figure 5.8. For example, `agg({3})` outputs the same sum of `3`, when `arr(a)<0` is mutated to `arr(a)>0`), because the loop starts from 1 instead of 0, and `sum` is initialize to the first element of the array. Thus, it is not possible to defect the fault using an array of length 1.

Exceptions. BIGTEST extends SPF to explicitly model exceptions. For example, when an expression involves a division operator, division by zero is possible, which can lead to program termination. In Figure 5.1, BIGTEST creates two additional terminating path conditions, due to division by zero (i.e., `x<y∧x==0` and `y≤x∧y==0`).

Combining UDF symbolic execution with equivalence classes. BIGTEST combines the path conditions of each UDF with the incoming constraints from its upstream operator. For example, the UDF of `filter` (❸) in Section 5.3 produces a path condition of `s._2 == "Palms"`. Suppose that the upstream operator `map` produces one non-terminating path condition `s.split(",").length≥2` with the effect `s._2 = splitn(s,",",1)`. Inside the equivalence classes of `filter`—rows ❶ and ❷ in Table 5.2, BIGTEST plugs in the incoming path conditions (/effects) of an upstream operator `map` to C_I and the path conditions (/effects) of the `filter`'s UDF to f, producing the following path conditions.

- $c(t) \wedge f(t)$: `s.split(",").length≥2 ∧ splitn (s,",",1) == "Palms"`
- $c(t) \wedge \neg f(t)$: `s.split(",").length≥2 ∧ ¬ (splitn (s,",",1) == "Palms")`

Joint Dataflow and UDF Path. BIGTEST defines the final set of paths of a big data application as *Joint Dataflow and UDF (JDU)* paths. We define a JDU path as follows: let $G = (D, E)$ represent a directed acyclic graph of a big data application where D is a set of vertices representing dataflow operators and E represents directed edges connecting dataflow operators. Imagine a big data application constructed with a `map` followed by `filter` and `reduce`. We represent this dataflow graph as $G = (D, E)$ such that $D = \{d_1, d_2, d_3, t_1\}$ and $E = \{(d_1, d_2), (d_2, d_3), (d_2, t_1)\}$ where d_1, d_2, and d_3 are `map`, `filter`, and `reduce` respectively. `filter` introduces a terminating edge (d_2, t_1) where a terminating vertex is t_1.

Since each dataflow operator takes a user-defined function f, for a vertex d_i, we define a subgraph $G_i = (V_i, E_i)$ which represents the control flow graph of f. In this subgraph, a vertex $v \in V_i$ represents a program point and an edge $(v_a, v_b) \in E_i$ represents the flow of control from v_a to v_b. G_i has $v_1 = start$ and $v_n = stop$ corresponding to the first and last statements. Then from each dataflow operator node d_i, we add a call edge from d_i to the start node of G_i and from the stop

```
(assert (= line2 (str.++ (str.++ line20 ",") line21)))
(assert
 (= line1
  (str.++ (str.++ " " ",")
   (str.++ (str.++ line11 ",")
    (str.++ (str.++ " " ",") (str.++ (str.++ line13 ",") line14))))))
(assert
  (and (not (= (str.to.int line14) 0))
   (and (isinteger line14)
    (and (isinteger line13)
     (and (= "Palms" line21)
      (and (= x11 line20)
       (and (<= s21 15)
        (and (<= s21 40) (and (= s21 x621) (and (= s1 x61) (= s22 x622))))))))))))))
(assert
  (and (= x11 line11)
   (and (= x12 (/ (str.to.int line13) (str.to.int line14)))
    (and (= x61 x11)
     (and (= x621 x12) (and (= x622 x42) (and (= x71 "walk") (= x72 1)))))))))))
```

Figure 5.9: Output SMT query constructed by BIGTEST to reflect JDU path constraint `C11` of Table 5.1 from motivating example.

node of G_i to the d_{i+1}. Since some UDFs include a loop and thus have a cycle in the control flow graph, we finitize the loop using a user provided bound K and unroll the loop K times.

We enumerate a set of all unique paths P_K for the graph G with expanded subgraphs and call each unique path a *Joint Dataflow and UDF (JDU) path*. For an arbitrary test suite T, the JDU path coverage is measured as a set of covered paths, $P_K(T) = \{p \mid p \in P_K,\ \exists t \in T \text{ and } t \models C_p\}$ where a test input t satisfies the path condition C_p of path p. Given a user-provided bound K for unrolling a loop, JDU path coverage is $\frac{|P_K(T)|}{|P_K|}$.

5.4.4 Test Data Generation

BIGTEST rewrites path constraints into an SMT query. For constraints on integer variables, BIGTEST uses analogous arithmetic and logical operators available in SMT. For string constraints, BIGTEST uses operations such as `str.++`, `str.to.int`, and `str.at`. BIGTEST introduces a new `splitn` symbolic operation. If a path constraint contains a clause `v = splitn("," s,1)`, BIGTEST generates `(assert (= s (str.++ " " (str.++ "," v))))` that is equivalent to `s = " ,`v`"` where v is a symbolic string. 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 generates interpreted functions for Java native methods not supported by Z3. For example, BIGTEST replaces `isInteger` with an analogous Z3 function. BIGTEST executes each

```
1  sc.textFile("hdfs://registrar:9999/gradebook.log")
2     .map { line => val arr = line.split(",")
3                    arr(1)
4     }
5     .map {   l => val a = l.split(":")
6                   (a(0), Integer.parseInt(a(1)))
7     }
8     .map {   a => if (a._2 > 40)
9                     ("Pass".concat(a._1), 1)
10                 else
11                    ("Fail".concat(a._1), 1)
12    }
13    .reduceByKey(_+_)
14    .filter ( v => v._2 <=2 && v._1.startsWith("Fail") )
```

Figure 5.10: A Spark program that identifies the courses with less than two failing students.

SMT query separately and finds satisfying assignments (i.e., test inputs) to exercise a particular path. While executing each SMT query independently may lead to redundant solving of overlapping constraints, in our experiments, we do not find it as a performance bottleneck. Theoretically, the number of path constraints increases exponentially due to branches and loops; however, empirically, our approach scales well to big data applications, because UDFs tend to be much smaller (in order of hundred lines) than DISC frameworks and we abstract the framework implementation using logical specifications.

Figure 5.9 shows an SMT query produced by BIGTEST for Figure 5.2. Lines 1 to 6 constrict the first table to have four segments and the second table to have two segments separated by a comma. Lines 7 to 10 restrict a string to be a valid integer. To enforce such constraint that crosses the boundary of strings and integers, BIGTEST uses a custom function `isinteger` and Z3 function `str.to.int`. Lines 11 to 14 enforce a record to contain "Palms" and the speed to be less than or equal to 15. Lines 15 to 19 join these constraints generated from a UDF to the subsequent dataflow operator.

5.5 Tool Interfaces

In this section, we present a step-by-step demonstration of BIGTEST. Suppose Alice writes a big data application in Apache Spark to find courses with fewer than two failing students. She uses the entire university gradebook database which contains several years of grading information spanning

```
1 filter1 = "",1
2 map3 = "",1
3 map4 = "CS:123"
4 reduceByKey2 = {1,2,3,4}
5 map5 = "a,a"
6 K_BOUND = 2
```

Figure 5.11: A configuration file for the motivating example

gigabytes. A sample row in this dataset contains comma-separated fields of a course id, a final mark in percentiles, year, a student id, a session name, and a major. Below is a small sample of the gradebook data.

```
CS233:77,1994,80554313,F1994,CS,..
CS233:53,1994,80594911,F1994,EE,..
CS233:29,1994,30472981,F1994,BIO,.
```

To perform this analysis, Alice writes a program, as shown in Figure 5.10. First, she loads the data from an HDFS storage using Apache Spark's `textFile` API in line 1. Once the data is loaded in Spark, she uses a `map` operator to extract the course id and a student's mark from each row using a UDF (lines 2 to 4). In the following `map` operation (lines 5 to 7), a string such as "CS233:77" is transformed into a tuple of a string and an integer. In lines 8 to 11, Alice annotates the tuples with either a "Pass" or a "Fail" string based on the marks. For each course, she calculates the total sum of passing and failing students using `reduceByKey` and then applies `filter` to find the courses with fewer than two failing students in line 14. This program has a total of 17 JDU paths, where 2 are non-terminating and 15 are terminating paths.

To run BIGTEST on her program, Alice first writes a configuration file in the format shown in Figure 5.11. She writes `K_BOUND=2` to limit the symbolic exploration of unbounded loops and collections to avoid path explosion. SPF requires sample input arguments to a function before symbolically executing it. BIGTEST takes such input arguments from Alice through `conf` file and passes them to SPF. A UDF can be identified with its corresponding operator name followed by the execution order number in reverse (similar to Spark's execution). Figure 5.11 shows the final configuration file. Alice invokes BIGTEST's command line tool using the following command.

```
java -jar BigTest.jar -enableBT /GradeAnalysis
```

Console Log 1

```
 Map4
Non-terminating:
PC: {l splitn 1 : isinteger && l = x2}
E: {x3_1 = l splitn 0 :, x3_2 = l splitn 1 : str.to.int }
Terminating:
PC: {l splitn 1 : notinteger} E: {}
PC: {l equals ""} E: {}
Map3
Non-terminating:
PC: {a_t2 <= 40 && a_t1 = x4_1 && a_t2 = x4_2}
E: {x5_1 = Fail str.++ a_t1, x5_2 = 1}
PC: {a_t2 > 40 && a_t1 = x4_1 && a_t2 = x4_2}
E: x5_1 = Pass str.++ a_t1, x5_2 = 1
```

Alice can monitor the progress of BIGTEST by looking at the console output log. BIGTEST logs progress with the number of paths explored, the path constraints, and the effects at each operator level until the final operator is reached. Console Log 1 shows the explored paths from the second and third map operator. `PC` refers to a path condition and `E` represents corresponding effect. For example, path constraint `l splitn 1 : notinteger` represents a terminating path emerging from `Integer.parseInt` at line 6 of Figure 5.10. Near the end of the execution, Alice starts to see the entire program's path constraints built on top of the individual path conditions of underlying operators. A sample console log snippet of a final non-terminating path constraint is shown below.

Console Log 2

```
PC: {l1 splitn 1 : isinteger && a2l1 <= 40 && l2 splitn 1 :
isinteger && a2l1 <= 40 && a select 1 = arr[1] 1 <
arr_length && a select 0 = arr[0] && x7 < 2 && x8
str.substr 0 4 equals Fail }
E: {x7 = arr[1] + arr[0] && lp1 = l1 splitn 0 && a2l1 = lp1 splitn 1 :
str.to.int && ...}
```

For every final path of the program, BIGTEST calls an SMT solver to produce a set of concrete input rows that Alice can use as a test data for her unit test.

Console Log 3

```
Path : 13
running CVC4 $>cvc4 --strings-exp --lang smt2 < /tmp/-966362206
line_1 () String ":41"
line_2 () String ":41"
line_3 () String ":0"
line_4 () String ":0"
```

Alice copies the test data generated by BIGTEST for path 13 (see Console Log 3) into a test data file and uses that file as input to a unit test. Due to small size, Alice comfortably runs this test on her local machine. Although the input does not contain any course with less than two failing students, the test output includes one row instead of zero. Upon further investigation, Alice identifies a code fault where she mistakenly used $\leq$ instead of $<$ (line 14 in Figure 5.10). Similarly, BIGTEST generates test inputs such as empty string or non-numeric string to reveal critical corner cases which can lead to a program crash at lines 3 and 6 in Figure 5.10. Alice fixes such cases through relevant exception handling and data filtering to eliminate the possibility of costly runtime crashes.

#	Subject Program	Output	# of Operators	Operators	Program Characteristics String Parsing	# Branches	# UDFs	JDU Paths (K=2)
P1	IncomeAggregate	Total income of individuals earning $\leq$\$300 weekly	3	map, filter, reduce	✓	2	4	6
P2	MovieRatings	Total number of movies with rating ≥ 4	4	map, filter, reduceByKey	✓	1	4	5
P3	AirportLayover	Total layer time of passengers per airport	3	map, filter, reduceByKey	✓	2	4	14
P4	CommuteType	Total number of people using each form of transport for daily commute	6	map, fitler, join, reduceByKey	✓	3	5	11
P5	PigMix-L2	PigMix performance benchmark	5	map, join	✓	2	6	4
P6	Grade Analysis	List of classes with more than 5 failing students	5	flatmap, filter, reduceByKey, map	✓	2	3	30
P7	WordCount	Finds the frequency of words	3	flatmap, map, reduceByKey	✓	1	3	4

Table 5.3: Subject Programs

5.6 Evaluation

We evaluate the effectiveness and efficiency of BIGTEST using a diverse set of benchmark big data applications. We compare BIGTEST against SEDGE in terms of path coverage, fault detection capability, and testing time. We compare test adequacy, input data size, and potential time saving against three alternative testing methods: (1) random sampling of $k\%$ records, and (2) using a subset of the first $k\%$ records, and (3) testing on the entire original data.

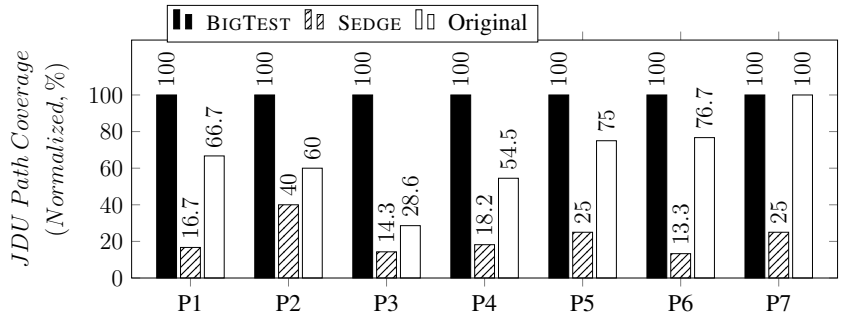


Figure 5.12: JDU path coverage of BIGTEST, SEDGE, and the original input dataset

- To what extent BIGTEST is applicable to big data applications?
- How much test coverage improvement can BIGTEST achieve?
- How many faults can BIGTEST detect?
- How much test data reduction does BIGTEST provide?
- How long does BIGTEST take to generate test data?

Subject Programs. In terms of benchmark programs, we use seven subject programs from earlier works on testing [91] and Chapter 4, listed in Table 5.3. The PigMix benchmark package contains a data generator script that generates large scale datasets. We utilize `map` and `flatmap` with UDFs in Apache Spark to translate unsupported Pig operators like `load As` and `split`. Three programs `MovieRating` (P2), `AirportLayover` (P3), and `WordCount` (P7) are adapted from BIGSIFT. Each program is paired with a large scale dataset. The rest are self-created custom Apache Spark applications to add heterogeneity in dataflow operators and UDFs. Table 5.3 shows detailed descriptions of subject programs. All applications (1) involve complex string operations including `split`, `substring`, and `toInt`, (2) perform complex arithmetics, (3) use type `Tuple` for key-value pairs, and (4) generate and process a collection with custom logic using `flatmap`.

Experimental Environment. We run all large-scale data processing on a 16-node cluster. Each node is running at 3.40GHz and equipped with 4 cores, 32GB of RAM, and 1TB of storage allowing us to run up to 120 tasks simultaneously. For storage, we use HDFS version 1.0.4 with a replication factor of 3. Due to a very small size of test data generated by BIGTEST, we leverage Apache Spark's local running mode to perform experiments on a single machine.

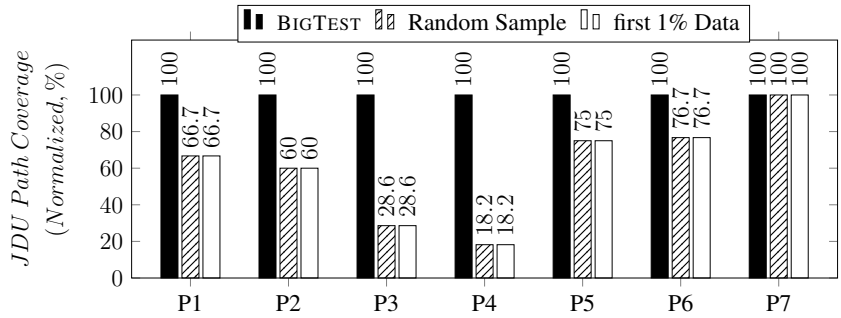


Figure 5.13: JDU path coverage of BIGTEST in comparison to alternative sampling methods

5.6.1 Big Data Application Support

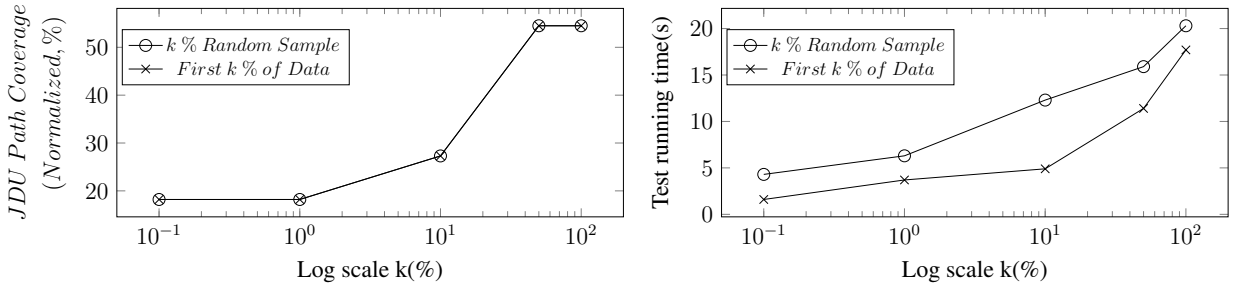
BIGTEST supports a variety of dataflow operators prevalent in big data applications. For instance, Apache Spark provides `flatmap` and `reduceByKey` for constructing and processing collections. The previous approach SEDGE is designed for PIG Latin with only a limited set of operators support [91]. SEDGE is neither open-source nor have any implementation available for Apache Spark for direct comparison. Therefore, we faithfully implement SEDGE precisely based on the technical details provided elsewhere [91]. We manually downgrade BIGTEST by removing symbolic execution for UDFs and equivalence classes for certain operators to emulate SEDGE. Out of seven benchmark applications written in Apache Spark, five applications contain `flatmap` and `reduceByKey`, therefore, SEDGE is not able to generate testing data for these 5 applications.

5.6.2 Joint Dataflow and UDF Path Coverage

We evaluate code coverage of BIGTEST, SEDGE, and the original input dataset based on JDU path coverage defined in Section 5.4.3.

JDU Path Coverage Evaluation. We compare BIGTEST with three alternative sampling techniques: (1) random sampling of $k\%$ of the original dataset, (2) selection of the first $k\%$ of the original dataset, as developers often test big data applications using `head -n`, and (3) a prior approach SEDGE. To keep consistency in our experiment setting, we enumerate JDU paths for a given user-provided bound K and measure how many of these paths are covered by each approach.

Figure 5.12 compares the test coverage from BIGTEST, SEDGE, and the original dataset. Y axis



(a) JDU Path Coverage

(b) Test Execution time

Figure 5.14: The number of JDU paths covered and the test execution time when $k\%$ of the data is randomly selected and the $first\ k\%$ of data is selected for subject program `CommuteType`.

represents the *normalized* JDU path coverage ranging from 0% to 100%. Across seven subject programs, we observe that SEDGE covers significantly fewer JDU paths (22% of what is covered by BIGTEST). By not modelling the internal paths of UDFs, SEDGE fails to explore many JDU paths. Even when the complete dataset is used, the JDU path coverage reaches only 66% of what BIGTEST could achieve. The entire dataset achieves better coverage than SEDGE but it still lacks coverage compared to BIGTEST. In other words, using the entire *big* data for testing does not necessarily provide high test adequacy.

In Figure 5.13, both *random 1% sample* and *first 1% sample* provide 59% of what is covered by BIGTEST. We perform another experiment to measure the impact of different sample sizes on JDU path coverage and test execution time. Figure 5.14a and Figure 5.14b present the results on `CommuteType`. In `CommuteType`, the covered JDU paths increases from two to six when the percentage of the selected data increases from 0.1% to 50%. For those small samples, input tables do not have matching keys to exercise downstream operators and the time and distance columns may not have specific values to exercise all internal paths of the UDF. In terms of running time, as the sample size (k) increases, the test execution time also increases linearly (see Figure 5.14b in which x-axis is in log scale).

5.6.3 Fault Detection Capability

We evaluate BIGTEST's ability to detect faults by manually injecting commonly occurring faults. Because big data applications are rarely open-sourced for data privacy reasons and there is no existing benchmark of faulty big data applications, we create a set of faulty big data applications by studying the characteristics of real world big data application bugs and injecting faults based on this study.

We carefully investigate Stack Overflow and Apache Spark Mailing lists with keywords; Apache Spark exceptions, task errors, failures, and wrong outputs and inspect top 50 posts. Many errors are related to performance and configuration errors; thus, we filter out those and analyze 23 posts related to coding errors. For each post, we investigate the type of fault by reading the question, posted code, error logs, answers, and accepted solutions. We categorize our findings into seven common fault types:

- incorrect string offset: *e.g.,* a user uses 1 instead of 0 as the starting index in method `substring` and encounters StringIndexOutOfBoundsException [11].
- incorrect column selection: *e.g.,* a user accesses a wrong column in a csv file and thus receives ArrayIndexOutOfBoundsException [9].
- use of wrong delimiters:*e.g.,* while splitting a string a user uses "[ ]" instead of "\[\]", leading to a wrong output [12].
- incorrect branch conditions: *e.g.,* a user places a wrong order of control predicates, executing only one branch's side [8].
- wrong join types: *e.g.,* a user uses a wrong relational operator such as `cartesian join` instead of inner `join` [7].
- swapping a key with a value: *e.g.,* a user tries to join two tables while the keys and values are interleaved [10].
- other common mutations such as incorrect arithmetic or Boolean operator in UDFs.

When applicable, we inject one of each fault type in every application. For example, fault types 1 and 3 could only be inserted when `substr` or `split` method is used. When a fault type is

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

Table 5.4: Fault detection capabilities of BIGTEST and SEDGE

applicable to multiple locations, we select a location which is inspired by and similar to the fault location in the corresponding StackOverflow/Mailing List post. For instance, for fault type (2) above, we manually modify code to extract the first column instead of the second as a key in line 4 of Figure 5.2. Similarly, for fault type (3), we introduce fault by replacing the delimiter ”,” with ”:”. In total, our benchmark comprises of 31 faulty big data applications. While SEDGE is not designed to handle string constraints, the main goal of this exercise is to justify the need to model UDFs and string constraints. SEDGE represents the internal UDFs as uninterpreted functions and, therefore, is unable to model all internal UDF paths. Conversely, BIGTEST treats UDFs as interpreted functions by representing them symbolically and models all internal UDF paths (up to bound k) which is crucial for high coverage testing of UDF's internal.

Table 5.4 shows a comparison of fault detection by BIGTEST and SEDGE. BIGTEST detects 2X more injected faults than SEDGE. For instance, in application P4, BIGTEST detects 6 faults, whereas SEDGE detects 4 faults. SEDGE uses concrete execution to model the UDF exercising line 16 of Figure 5.2 only. Therefore, it is unable to find an input for detecting fault at line 17 when the binary operator `">"` is replaced with `"<"` (*i.e.,* `s._2._1>15` to `s._2._1<15`). Similarly, when `join` in line 13 is changed to `rightOuterJoin`, SEDGE cannot detect any difference in the output because the equivalence classes do not model the terminating cases of join.

Approach	Test Input Data		Output from program	
			Original	Faulty
BIGTEST	Terminating	CS100:41,0	CS200	CS100
	Non-terminating	CS200:0,0,0,0,0,0		CS200
Alternative	Non-terminating	CS200:0,0,0,0,0,0	CS200	CS200

Table 5.5: Modelling terminating and non-terminating cases

As another example, application P6 identifies courses with more than 5 failing students. A

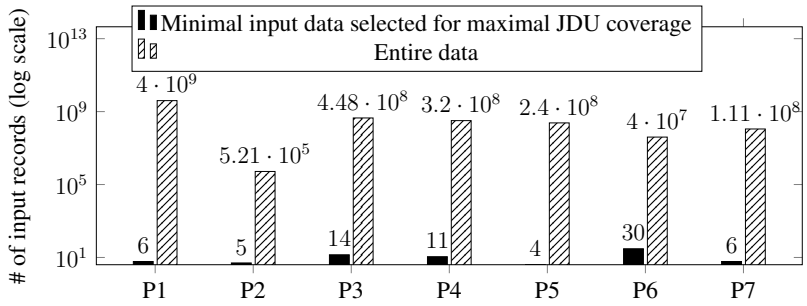


Figure 5.15: Reduction in the size of the testing data by BIGTEST

faulty version of P6 replaces the filter predicate `count>5` to `count>0` to output courses with at least one failing student. The original version of P6 uses `map` and `filter` to parse each row and identify failing students, `reduceByKey` to count the number of failing students, and uses `filter` to find courses with more than 5 failing students. BIGTEST generates at least two records to exercise both terminating and non-terminating cases of the last `filter`; thus, the original and faulty versions produce different outcomes on this data. On the other hand, a record is generated to exercise a non-terminating case only. Such data would produce the same outcome for both the original and the faulty versions, unable to detect the injected fault, as shown in Table 5.5.

5.6.4 Testing Data Reduction

Testing big data applications on the entire dataset is expensive and time-consuming. BIGTEST minimizes the size of the dataset, while maintaining the same test coverage. It generates only a few data records (in order of tens) to achieve the same JDU path coverage as the entire production data. Four out of seven benchmarks have an accompanied dataset, whereas the rest relies on a synthetic dataset of around 20GB each. Figure 5.15 shows the comparison result. In application P6, BIGTEST generates 30 rows of data to achieve 33% more JDU path coverage than the entire dataset of 40 million records. In other words, BIGTEST produces testing data 10^6 times smaller than the original dataset. Across all benchmark applications, BIGTEST generates data ranging from 5 to 30 rows. This is 10^5 to 10^8 times smaller than the original dataset, showing the potential to significantly reduce dataset size for local testing.

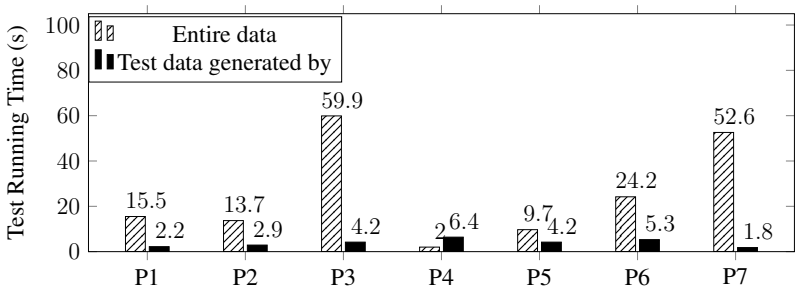


Figure 5.16: Test running time of entire data on large-scale cluster vs. testing on local machine with BIGTEST

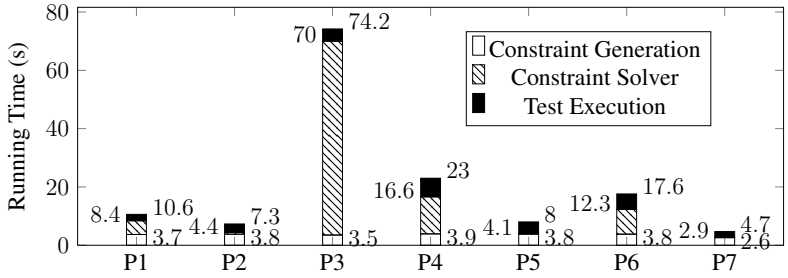


Figure 5.17: Breakdown of BIGTEST's running time

5.6.5 Time and Resource Saving

By minimizing test data without compromising JDU path coverage, BIGTEST consequently reduces the test running time. The benefit of a smaller test data is twofolds: (1) the amount of time required to run a test case decreases, and (2) the amount of resources (worker nodes, memory, disk space, etc.) for running tests also decreases.

We measure, on a single machine, the total running time by BIGTEST and compare it with the testing time on a 16-node cluster with the entire input dataset. We present a breakdown of the total running time into test data generation vs. executing an application on the generated data. Figure 5.16 represents the evaluation results. In application P6, it takes 5.3 seconds on a single machine to test with data from BIGTEST otherwise testing takes 387.2 CPU seconds (24.2 seconds x 16 machines) on the entire dataset, which still lacks complete JDU path coverage. Across the seven subject programs, BIGTEST improves the testing time by 194X, on average, compared to testing with the entire dataset.

Figure 5.17 reports the complete breakdown of the total running time of BIGTEST. The max-

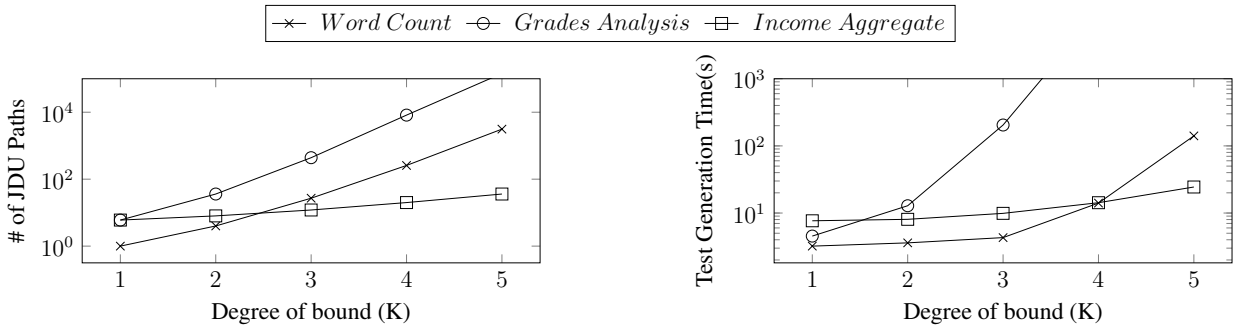


Figure 5.18: BIGTEST's performance when the degree of upper bound (K) on loop iteration and collection size changes

imum test generation time observed is 70 seconds for `Airport Layover` (P3) in which 66 seconds are consumed by constraint solving. This is because the resulting JDU paths include integer arithmetics and complex string constraints together. Solving such constraints that cross the boundaries of different dimensions (integer arithmetics vs. string constraints) is time consuming even after BIGTEST's optimizations. If we combine both the test running time and test generation time and compare BIGTEST with the testing time with the entire dataset, BIGTEST still outperforms. In fact, BIGTEST still is 59X faster than testing on the entire dataset.

5.6.6 Bounded Depth Exploration

BIGTEST takes a user-provided bound K to bound the number of times a loop is unrolled. We assess the impact of varying K from 1 to 5 and present the results in Figure 5.18. At K=2, the number of JDU paths for `GradeAnalysis` is 36. When K is 3, BIGTEST generates 438 JDU paths. An exponential-like increase in the test generation time can be seen across the subject program, as we increase K. When K=2 in `GradeAnalysis`, BIGTEST takes 12 seconds and with K=3, BIGTEST takes 204 seconds. We empirically find K=2 to be a reasonable upper bound for loop iteration to avoid path explosion.

5.6.7 Threats to Validity

As we manually seed faults in the benchmark applications, the location of faults may introduce a bias in fault detection rate of BIGTEST posing a threat to internal validity. However, as mentioned before, most type of faults are only applicable to a single code location. If a fault type is applicable to multiple locations, we then select the fault location inspired by the corresponding StackOverflow/Mailing List post. In case of external validity, our classification of DISC faults may not be representative of all possible big data application faults out there, as the survey is based on 50 StackOverflow/mailing lists posts. Additionally, the selection of fault types in our evaluation may be unfair to prior approaches. We attempt to mitigate this bias by restricting the evaluation to top seven most commonly occurring faults in big data applications. To eliminate this threat in the future, we plan to perform a large scale study on big data application faults.

5.7 Summary

To enable efficient and effective testing of big data analytics in real world settings, this chapter presents a symbolic execution based white-box testing technique that systematically explores the *combined* behavior of dataflow operators and corresponding UDFs. Our evaluation shows that BIGTEST can detect 2X more faults than the previous approach and can consume 194X less CPU time, on average than using the entire dataset. With BIGTEST, fast local testing is feasible and testing big data applications on the entire dataset may not be necessary.

CHAPTER 6

Conclusion

6.1 Summary

Due to their widespread usage, DISC frameworks have achieved remarkable performance milestones and have revolutionized the domain of big data analytics. Unfortunately, testing and debugging for big data analytics is currently an ad-hoc and time-consuming process, mostly involving guesswork—developers often resort to *"random sample-based"* testing and *"printf based trial and error"* debugging.

This dissertation investigates the feasibility of established and provably-useful testing and debugging concepts from software engineering for the domain of big data analytics by combining ideas from big data systems and databases. To this end, we address three principal research questions: (1) *what are the necessary debugging primitives for interactive big data analytics?*, (2) *how can we isolate the root cause of a crash, failure, or suspicious output?* and (3) *how can we test a big data application efficiently and effectively?*. As a first step towards debugging, we demonstrate with BIGDEBUG that by leveraging the deterministic and immutable nature of big data applications, we can enable effective and interactive debugging primitives that pose minimal performance overhead. We further investigate how to automate the debugging process and conclude that by tailoring data provenance for the purpose of delta debugging, our tool BIGSIFT can automatically identify the most concise fault-inducing input data in just under the original job execution time. Finally, we demonstrate that by abstracting the implementation of dataflow operations in DISC frameworks, we can generate the most concise synthetic test data that is capable of revealing defects that *testing on the entire input data* often overlooks. In a nutshell, this dissertation validates that by design-

ing interactive and automated debugging and testing techniques that are specifically customized towards big data analytics running on DISC framework, we can reduce time for debugging with minimal performance overhead and reduce the number of inputs required for testing.

We believe that there are more opportunities for adapting existing software engineering methods for big data analytics. In the following sections, we lay out a plan to explore these new research areas and discuss corresponding hypotheses.

6.2 Future Research Directions

Influence Based Debugging of Big Data Analytics. Debugging big data analytics often involves asking a "Why" question—Why does the program produce a specific output? Database research addresses this question by leveraging data provenance that isolates the inputs records responsible for generating an output record under investigation. While capturing data lineage at runtime, existing data provenance techniques assign equal weights to all inputs records that contribute to producing a specific output. However, in reality, the magnitude of the contribution (also know as *influence*) is non-uniform among the input records that are part of the output data lineage. Based on this insight, we ask the following research question: *"Among the data lineage of an output record, which input has the biggest influence on the output?"*.

With my colleagues at UCLA, I am currently investigating two preliminary hypotheses that combine insights from software engineering, database systems, and differential calculus.

1. Using a pre- or user-defined influence function for every aggregate operator in a big data application, we can rank the input records based on their contribution towards a final aggregated output.

2. By representing a program in a symbolic mathematical expression, we can leverage first-order differential calculus to generate an influence function automatically.

For the first hypothesis, my colleagues and I propose a data provenance algorithm that redesigns

aggregation operators in Apache Spark, such as `reduceByKey` and `aggregateByKey` to leverage an influence function to rank the inputs based on their contribution towards the final output. We plan to provide a set of pre-defined influence functions in addition to user-defined influence function. Furthermore, to increase the precision of our data provenance approach, we must track which code paths are exercised by individual records inside UDFs. For example, suppose that a UDF `list => return list.head()` is used within a `map` transformation that only returns the first element of a list. Therefore, the output lineage should contain only one element of the list. We plan to perform a white-box taint analysis on user-defined functions (UDFs) to capture these precise input-output mappings within a UDF. Our preliminary evaluations validate our hypothesis and show promising improvements towards improving the precision of data provenance and reducing the runtime storage cost of recording data provenance.

For the second hypothesis, we get inspiration from a fundamental theory in differential calculus *i.e.,* if an input maximizes the derivative of a mathematical expression, a slight change to that input will have the highest impact on output. Using this insight, we propose to transform a big data application into a set of symbolic expressions that can be differentiated based on custom derivative rules to construct an influence function. A challenge in this approach is to find an approximate symbolic representation of *" point of discontinuity" i.e.,* branches, loops, and dataflow operators. We previously targeted a similar research problem in BIGSIFT (Chapter 4), which is a post-mortem debugging technique. In contrast, the proposed approaches are more generic, promises higher accuracy and precision, and poses minimal runtime overhead.

White-Box Performance Modeling of Big Data Applications. Big data applications are usually developed and tested on a single machine. However, a variety of factors such as data partitioning, network speed, and worker memory can impact the performance of an application on the cluster. This unpredictability leads to frustration when the application is unexpectedly slow on the cluster. Due to the lack of visibility into the execution, users often preemptively terminate the application under the assumption that something went wrong, wasting many hours of computation as well as impeding developer productivity. Since the running time of a big data application can be anywhere

between several minutes to several days, we ask an important research question: *"What would be the expected running time of a big data application?"*. To address this question, we must perform program analysis with software engineering techniques and combine it with the execution model of DISC systems to extract the factors that affect the performance of a big data application.

Prior studies of performance bottlenecks in big data workloads [110] report CPU, network, and data skew as the primary suspects. We experience similar behavior in our past research where the execution time of a big data application is highly influenced by (1) the framework's configuration such as CPUs, executor memory, and number of workers nodes, (2) the physical plan of an application, and (3) the input data distribution. Related work on performance prediction such as Ernest [134] does not incorporate the data distribution and other program semantics, leading to low prediction accuracy. Based on these insights, we draw the following hypothesis: *Based on the physical plan, the framework's configuration, and the input dataset, we can estimate the relative running time of a big data application.* To validate this hypothesis, we propose a statically-generated performance prediction model that incorporates the physical plan of the big data application, the framework's configuration, and the partitioning scheme of the input dataset. Based on the testing runs, we plan to tune the parameters of our prediction model and then plug in the data partition scheme and DISC configuration to predict an approximate running time. We plan to evaluate this hypothesis on a series of big data benchmark (*e.g.,* PUMA [15]) as well as performance workloads (*e.g.,* TPC [130]). With the knowledge of the expected running time of an application, a user can better anticipate the application performance and differentiate between performance bug and variability.

Debugging Big Data Applications Running on Heterogeneous Environment. Hardware acceleration has shown great promises in overcoming the performance barrier of traditional CPU-based computing. FPGAs and GPUs are the two most popular hardware extensions that can help enhance the performance of DISC systems. Prior work on utilizing such hardware in Apache Spark demonstrate the utility of such heterogeneous frameworks without requiring too much manual effort to transform big data applications [37, 143]. However, debugging is yet to be fully supported in such

frameworks. Therefore, we ask the following research question:*"how can we perform debugging for big data applications running on heterogeneous computing environments?"*. DISC frameworks that leverage heterogeneous computing include a resource manager that schedules computation tasks on either a CPU or an FPGA. In case of a failure or a crash, we plan to re-design the resource managers to maintain a scheduling log, containing the input data partition required to reproduce the error on the CPU. Moreover, we plan to get inspiration from our automated debugging tool (Chapter 4) and accelerate the iterative debugging workloads on FPGA to isolate the minimum input within the failure data partition.

Based on our preliminary idea, our hypothesis is: *By keeping a minimum log of task schedules in heterogeneous computing, we can reproduce runtime failures on CPU and quickly isolate the precise faulty data by running debugging workload on accelerated hardware.* We will extend existing frameworks such as Blaze [78] and Spark-GPU [143] to build research prototypes and evaluate them on commonly available iterative workloads such as TPC-H benchmarks [130], genomics sequencing, K-Means, and linear regression. To showcase the debugging capabilities, we will inject commonly occurring data or code faults in our benchmarks programs and evaluate our tool's precision and accuracy in detecting injected defects. With such debugging techniques, we can contribute towards improving the development experience of big data applications running on accelerated hardware and hence, lower the barrier to entry.

Configuration Debugging of Big Data Applications. DISC frameworks expose a plethora of configuration knobs designed to be tuned based on the cluster setting (*e.g.,* number of workers), input data, and user applications. However, novice users cannot gauge the impact of these configurations, which leads to under or overutilization of resources. A preliminary study conducted by my colleague at UCLA shows that 64% of the total StackOverflow posts are related to framework configuration tuning. An extensive scale survey conducted by Bagherzadeh et al. verifies that configuration, storage, and performance topics collectively constitute 39% of all big data topics on StackOverflow [24]. Both studies motivate us to facilitate configuration tuning and debugging and pose the following research question: *"how can we model the impact of each configuration knob*

on the runtime behavior of an application?". In order to explore this question, we plan to leverage software engineering techniques such as symbolic execution to understand program semantics and configuration specifications derived from big data systems to model the impact of commonly used configuration parameters. For example, if the entire input dataset contains only four unique keys, then, `groupByKey` and succeeding operators can only benefit from a maximum of four worker nodes in a DISC cluster.

Our hypothesis is: *By leveraging the specifications of configurations in the DISC system and the semantics of big data application, we can identify the precise configuration parameter responsible for generating runtime issues such as non-responsive performance, heap space error, and data skew*. We plan to evaluate our hypothesis on popular configurations issues reported on StackOverflow and compare our results with the correct answers reported through manual debugging. As a first step towards investigating this hypothesis, my colleague and I propose a data provenance based performance debugging approach that combines record-level data lineage with the computational latency of individual input record [126]. Using this algorithm, we can precisely identify the most *expensive* set of input records—records that have the most significant impact on the execution time of a big data application.

Developer productivity has been a key area of research in software engineering in the past. As we move into the future, it is essential to support the increasing population of big data application developers with the right productivity tools. In this dissertation, we aspire to get inspiration from multiple research areas and mold the research questions targeting developer productivity into performance and scalability questions under big data systems and databases. By doing so, we open new perspectives of looking at problems with a different outlook and get the *"the best of both worlds"*. As the data-centric world expands, the challenges in this domain will get more intricate. So far, this dissertation has bridged the gap between big data systems and software engineering, and in the future, we plan to research impactful problems in software engineering through the lens of experts from other disciplines.

REFERENCES

[1] Amazon s3. https://aws.amazon.com/s3/.

[2] Bigtest faulty big data applications benchmark. https://github.com/maligulzar/BigTest.

[3] Hadoop. http://hadoop.apache.org/.

[4] Mllib. http://spark.apache.org/mllib/.

[5] Scala.tool.nsc. http://www.scala-lang.org/api/2.11.0/scala-compiler/index.html#scala.tools.nsc.package.

[6] Spark documentation. http://spark.apache.org/docs/1.2.1/.

[7] 2015. https://stackoverflow.com/questions/32190828.

[8] 2016. https://stackoverflow.com/questions/40494999.

[9] 2017. https://stackoverflow.com/questions/48021303.

[10] 2017. https://stackoverflow.com/questions/42459749.

[11] 2018. https://stackoverflow.com/questions/49505241.

[12] 2018. https://stackoverflow.com/questions/52083828.

[13] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI 90, page 246256, New York, NY, USA, 1990. Association for Computing Machinery.

[14] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP 03, page 7489, New York, NY, USA, 2003. Association for Computing Machinery.

[15] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar. Puma: Purdue mapreduce benchmarks suite. 2012.

[16] G. Altekar and I. Stoica. Dcr: Replay debugging for the datacenter. Technical Report UCB/EECS-2010-74, EECS Department, University of California, Berkeley, May 2010.

[17] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: Exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys 10, page 223236, New York, NY, USA, 2010. Association for Computing Machinery.

[18] P. Alvaro and S. Tymon. Abstracting the geniuses away from failure testing. *Queue*, 15(5):2953, Oct. 2017.

[19] M. K. Anand, S. Bowers, and B. Ludscher. Provenance browser: Displaying and querying scientific workflow provenance graphs. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 1201–1204, 2010.

[20] K. M. Anderson. Embrace the challenges: Software engineering in a big data world. In *2015 IEEE/ACM 1st International Workshop on Big Data Software Engineering*, pages 19–25, 2015.

[21] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[22] A. Aydin, L. Bang, and T. Bultan. Automata-based model counting for string constraints. In D. Kroening and C. S. Păsăreanu, editors, *Computer Aided Verification*, pages 255–272, Cham, 2015. Springer International Publishing.

[23] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA 11, page 1222, New York, NY, USA, 2011. Association for Computing Machinery.

[24] M. Bagherzadeh and R. Khatchadourian. Going big: A large-scale study on what big data developers ask. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*,

ESEC/FSE 2019, page 432442, New York, NY, USA, 2019. Association for Computing Machinery.

[25] L. Bang, A. Aydin, Q.-S. Phan, C. S. Păsăreanu, and T. Bultan. String analysis for side channels with segmented oracles. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 193–204, New York, NY, USA, 2016. ACM.

[26] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.

[27] O. Biton, S. Cohen-Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE 08, page 10721081, USA, 2008. IEEE Computer Society.

[28] R. S. Boyer, B. Elspas, and K. N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM.

[29] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[30] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI08, page 209224, USA, 2008. USENIX Association.

[31] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International Conference on Model Checking Software*, SPIN'05, pages 2–23, Berlin, Heidelberg, 2005. Springer-Verlag.

[32] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[33] M. Carbin and M. C. Rinard. Automatically identifying critical input regions and code in applications. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA 10, page 3748, New York, NY, USA, 2010. Association for Computing Machinery.

[34] R. Chaiken, B. Jenkins, P.-r. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):12651276, Aug. 2008.

[35] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan. Flume-java: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010.

[36] T. W. Chan and A. Lakhotia. Debugging program failure exhibited by voluminous data. *Journal of Software Maintenance*, 10:111–150, 1998.

[37] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, and P. Wei. When apache spark meets fpgas: A case study for next-generation dna sequencing acceleration. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud16, page 6470, USA, 2016. USENIX Association.

[38] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 02, page 210220, New York, NY, USA, 2002. Association for Computing Machinery.

[39] Z. Chothia, J. Liagouris, F. McSherry, and T. Roscoe. Explaining outputs in modern data analytics. *Proc. VLDB Endow.*, 9(12):11371148, Aug. 2016.

[40] S. Chu, C. Wang, K. Weitz, and A. Cheung. Cosette: An automated prover for sql. In *CIDR*, 2017.

[41] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA 07, page 196206, New York, NY, USA, 2007. Association for Computing Machinery.

[42] J. Clause and A. Orso. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA 09, page 249260, New York, NY, USA, 2009. Association for Computing Machinery.

[43] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 342–351, 2005.

[44] Z. Coker, D. G. Widder, C. Le Goues, C. Bogart, and J. Sunshine. A qualitative study on framework debugging. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 568–579, 2019.

[45] B. Contreras-Rojas, J.-A. Quiané-Ruiz, Z. Kaoudi, and S. Thirumuruganathan. Tagsniff: Simplified big data debugging for dataflow jobs. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC 19, page 453464, New York, NY, USA, 2019. Association for Computing Machinery.

[46] C. Csallner, L. Fegaras, and C. Li. New ideas track: Testing mapreduce-style programs. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 504–507, New York, NY, USA, 2011. ACM.

[47] Y. Cui. *Lineage Tracing in Data Warehouses*. PhD thesis, Stanford, CA, USA, 2002. AAI3038081.

[48] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179227, June 2000.

[49] D. Dao, J. Albrecht, C. Killian, and A. Vahdat. Live debugging of distributed systems. In O. de Moor and M. I. Schwartzbach, editors, *Compiler Construction*, pages 94–108, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[50] A. Dave, M. Zaharia, S. Shenker, and I. Stoica. Arthur: Rich post-facto debugging for production analytics applications.

[51] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[52] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[53] R. DeLine. Modern software is all about data. development environments should be, too. In *SPLASH Companion 2015 Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanit*, page 4. ACM, October 2015.

[54] R. DeLine. Research opportunities for the big data era of software engineering. In *Big Data Software Engineering (BIGDSE), 2015 IEEE/ACM 1st International Workshop on*. IEEE, May 2015.

[55] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 102–112, New York, NY, USA, 2000. ACM.

[56] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 151–162, New York, NY, USA, 2007. ACM.

[57] D. Fisher, R. DeLine, M. Czerwinski, and S. Drucker. Interactions with big data analytics. *Interactions*, 19(3):5059, May 2012.

[58] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *International conference on Data Mining (full paper)*. IEEE, December 2009.

[59] S. Gathani, P. Lim, and L. Battle. Debugging database queries: A survey of tools, tech-

niques, and users. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI 20, page 116, New York, NY, USA, 2020. Association for Computing Machinery.

[60] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design Implementation*, NSDI07, page 21, USA, 2007. USENIX Association.

[61] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[62] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. *Proceedings of OSDI*, pages 599–613, 2014.

[63] M. Gulzar, Y. Zhu, and X. Han. Perception and practices of differential testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 71–80. IEEE, 2019.

[64] M. A. Gulzar, M. Interlandi, T. Condie, and M. Kim. Bigdebug: Interactive debugger for big data analytics in apache spark. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 10331037, New York, NY, USA, 2016. Association for Computing Machinery.

[65] M. A. Gulzar, M. Interlandi, X. Han, M. Li, T. Condie, and M. Kim. Automated debugging in data-intensive scalable computing. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC 17, page 520534, New York, NY, USA, 2017. Association for Computing Machinery.

[66] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein, and M. Kim. Bigdebug: Debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE 16, page 784795, New York, NY, USA, 2016. Association for Computing Machinery.

[67] M. A. Gulzar, S. Mardani, M. Musuvathi, and M. Kim. White-box testing of big data analytics with complex user-defined functions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 290301, New York, NY, USA, 2019. Association for Computing Machinery.

[68] M. A. Gulzar, S. Wang, and M. Kim. Bigsift: Automated debugging of big data analytics in data-intensive scalable computing. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 863866, New York, NY, USA, 2018. Association for Computing Machinery.

[69] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC 16, page 116, New York, NY, USA, 2016. Association for Computing Machinery.

[70] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. OSDI, January 2008.

[71] B. P. Gupta, D. Vira, and S. Sudarshan. X-data: Generating test data for killing sql mutants. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 876–879, 2010.

[72] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE 05, page 263272, New York, NY, USA, 2005. Association for Computing Machinery.

[73] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 312–326, New York, NY, USA, 2001. ACM.

[74] T. Heinis and G. Alonso. Efficient lineage tracking for scientific workflows. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD 08, page 10071018, New York, NY, USA, 2008. Association for Computing Machinery.

[75] M. Herschel, R. Diestelkämper, and H. Ben Lahmar. A survey on provenance: What for? what form? what from? *The VLDB Journal*, 26(6):881–906, 2017.

[76] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE 15, page 483493. IEEE Press, 2015.

[77] W. E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Trans. Softw. Eng.*, 3(4):266–278, July 1977.

[78] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong. Programming and runtime support to blaze fpga accelerator deployment at datacenter scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC 16, page 456469, New York, NY, USA, 2016. Association for Computing Machinery.

[79] O. Hummel, H. Eichelberger, A. Giloj, D. Werle, and K. Schmid. A collection of software engineering challenges for big data system development. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 362–369. IEEE, 2018.

[80] R. Ikeda, J. Cho, C. Fang, S. Salihoglu, S. Torikai, and J. Widom. Provenance-based debugging and drill-down in data-oriented workflows. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1249–1252, 2012.

[81] R. Ikeda, J. Widom, and A. Das Sarma. Logical provenance in data-oriented workflows? In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE 13, page 877888, USA, 2013. IEEE Computer Society.

[82] M. Interlandi, A. Ekmekji, K. Shah, M. A. Gulzar, S. D. Tetali, M. Kim, T. Millstein, and T. Condie. Adding data provenance support to apache spark. *The VLDB Journal*, 27(5):595615, Oct. 2018.

[83] V. Jagannath, Z. Yin, and M. Budiu. Monitoring and debugging dryadlinq applications with daphne. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1266–1273, 2011.

[84] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck, and P. Buono. Research directions in data wrangling: Visualizations and transformations for usable and credible data. *Information Visualization*, 10(4):271–288, 2011.

[85] M. G. Kang, S. McCamant, P. Poosankam, and D. X. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[86] M. Kim, T. Zimmermann, R. DeLine, and A. Begel. The emerging role of data scientists on software development teams. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE 16, page 96107, New York, NY, USA, 2016. Association for Computing Machinery.

[87] M. Kim, T. Zimmermann, R. DeLine, and A. Begel. Data scientists in software teams: State of the art and challenges. *IEEE Transactions on Software Engineering*, 44(11):1024–1038, 2018.

[88] T. Kraska. Northstar: An interactive data science system. *Proc. VLDB Endow.*, 11(12):21502164, Aug. 2018.

[89] T. Leblanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. Apr. 1987.

[90] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Softw. Pract. Exper.*, 46(11):1547–1569, Nov. 2016.

[91] K. Li, C. Reichenbach, Y. Smaragdakis, Y. Diao, and C. Csallner. Sedge: Symbolic example data generation for dataflow programs. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE13, page 235245. IEEE Press, 2013.

[92] N. Li, Y. Lei, H. R. Khan, J. Liu, and Y. Guo. Applying combinatorial test data generation

to big data applications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 637–647, 2016.

[93] S. Li, H. Zhou, H. Lin, T. Xiao, H. Lin, W. Lin, and T. Xie. A characteristic study on failures of production distributed data-parallel programs. In *ICSE (SEIP track). Best paper!* ACM/IEEE, May 2013. 2013 IEEE Software Best Software Engineering in Practice Paper Award.

[94] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages Applications*, OOPSLA 13, page 1932, New York, NY, USA, 2013. Association for Computing Machinery.

[95] J. Lin and D. Ryaboy. Scaling big data mining infrastructure: The twitter experience. *SIGKDD Explor. Newsl.*, 14(2):619, Apr. 2013.

[96] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3s: Debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI08, page 423437, USA, 2008. USENIX Association.

[97] D. Logothetis, S. De, and K. Yocum. Scalable lineage capture for debugging disc analytics. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC 13, New York, NY, USA, 2013. Association for Computing Machinery.

[98] S. Ma, Y. Aafer, Z. Xu, W.-C. Lee, J. Zhai, Y. Liu, and X. Zhang. Lamp: Data provenance for graph based machine learning algorithms through derivative computation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 786797, New York, NY, USA, 2017. Association for Computing Machinery.

[99] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[100] F. McSherry, D. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Proceedings of CIDR 2013*, January 2013.

[101] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proc. VLDB Endow.*, 4(1):3445, Oct. 2010.

[102] Z. Miao, S. Roy, and J. Yang. Explaining wrong queries using small examples. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD 19, page 503520, New York, NY, USA, 2019. Association for Computing Machinery.

[103] G. Misherghi and Z. Su. Hdd: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE 06, page 142151, New York, NY, USA, 2006. Association for Computing Machinery.

[104] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 67–82, Berkeley, CA, USA, 2009. USENIX Association.

[105] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. *The Journal of Supercomputing*, 8(4):371–388, 1995.

[106] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. 02 2005.

[107] C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 245–256, New York, NY, USA, 2009. ACM.

[108] C. Olston and B. Reed. 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*, SIGMOD 11, page 12211224, New York, NY, USA, 2011. Association for Computing Machinery.

[109] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International*

Conference on Management of Data, SIGMOD 08, page 10991110, New York, NY, USA, 2008. Association for Computing Machinery.

[110] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI15, page 293307, USA, 2015. USENIX Association.

[111] K. Pan, X. Wu, and T. Xie. Database state generation via dynamic symbolic execution for coverage criteria. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, DBTest '11, pages 4:1–4:6, New York, NY, USA, 2011. ACM.

[112] H. Park, R. Ikeda, and J. Widom. Ramp: A system for capturing and tracing provenance in mapreduce workflows. In *37th International Conference on Very Large Data Bases (VLDB)*. Stanford InfoLab, August 2011.

[113] F. Psallidas and E. Wu. Smoke: Fine-grained lineage at interactive speed. *Proc. VLDB Endow.*, 11(6):719732, Feb. 2018.

[114] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. 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, pages 15–26, New York, NY, USA, 2008. ACM.

[115] A. Rabkin and R. Katz. Chukwa: A system for reliable large-scale log collection. In *Proceedings of the 24th International Conference on Large Installation System Administration*, LISA10, page 115, USA, 2010. USENIX Association.

[116] C. V. Ramamoorthy, S. B. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Trans. Softw. Eng.*, 2(4):293–300, July 1976.

[117] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, Aug. 1996.

[118] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of*

Data, SIGMOD 14, page 15791590, New York, NY, USA, 2014. Association for Computing Machinery.

[119] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP 10, page 317331, USA, 2010. IEEE Computer Society.

[120] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, page 263272, New York, NY, USA, 2005. Association for Computing Machinery.

[121] J. Shafer, S. Rixner, and A. L. Cox. The hadoop distributed filesystem: Balancing portability and performance. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 122–133. IEEE, 2010.

[122] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 402–411, 2013.

[123] M. Staats and C. Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 183–194, New York, NY, USA, 2010. ACM.

[124] R. Sumbaly, J. Kreps, and S. Shah. The big data ecosystem at linkedin. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD 13, page 11251134, New York, NY, USA, 2013. Association for Computing Machinery.

[125] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. Salsa: Analyzing logs as state machines. In *Proceedings of the First USENIX Conference on Analysis of System Logs*, WASL08, page 6, USA, 2008. USENIX Association.

[126] J. Teoh, M. A. Gulzar, G. H. Xu, and M. Kim. Perfdebug: Performance debugging of computation skew in dataflow systems. In *Proceedings of the ACM Symposium on Cloud*

Computing, SoCC 19, page 465476, New York, NY, USA, 2019. Association for Computing Machinery.

[127] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD 10, page 10131020, New York, NY, USA, 2010. Association for Computing Machinery.

[128] N. Tillmann and P. de Halleux. Pex - white box test generation for .net. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153. Springer Verlag, April 2008.

[129] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 14, page 530541, New York, NY, USA, 2014. Association for Computing Machinery.

[130] T. P. P. C. TPC. Tpc benchmark ds. 2015.

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

[132] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni. Automated decomposition of build targets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 123–133, 2015.

[133] M. Veanes, J. d. Halleux, N. Tillmann, and P. de Halleux. Qex: Symbolic sql query explorer. Technical Report MSR-TR-2009-2015, October 2009. Updated January 2010.

[134] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI16, page 363378, USA, 2016. USENIX Association.

[135] W. Visser, C. S. Pundefinedsundefinedreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on*

Software Testing and Analysis, ISSTA 04, page 97107, New York, NY, USA, 2004. Association for Computing Machinery.

[136] X. Wang, X. L. Dong, and A. Meliou. Data x-ray: A diagnostic tool for data errors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD 15, page 12311245, New York, NY, USA, 2015. Association for Computing Machinery.

[137] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE 81, page 439449. IEEE Press, 1981.

[138] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *Proc. VLDB Endow.*, 6(8):553564, June 2013.

[139] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP 09, page 117132, New York, NY, USA, 2009. Association for Computing Machinery.

[140] Z. Xu, M. Hirzel, G. Rothermel, and K. L. Wu. Testing properties of dataflow program operators. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 103–113, Nov 2013.

[141] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI08, page 114, USA, 2008. USENIX Association.

[142] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, Broomfield, CO, Oct. 2014. USENIX Association.

[143] Y. Yuan, M. F. Salmi, Y. Huai, K. Wang, R. Lee, and X. Zhang. Spark-gpu: An accelerated

in-memory data processing engine on clusters. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 273–283, 2016.

[144] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. 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*, NSDI12, page 2, USA, 2012. USENIX Association.

[145] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud10, page 10, USA, 2010. USENIX Association.

[146] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT 02/FSE-10, page 110, New York, NY, USA, 2002. Association for Computing Machinery.

[147] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183200, Feb. 2002.

[148] H. Zhou, J.-G. Lou, H. Zhang, H. Lin, H. Lin, and T. Qin. An empirical study on quality issues of production big data platform. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE 15, page 1726. IEEE Press, 2015.