

Automated Testing for Data-Intensive Scalable Computing

Ahmad Humayun

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science & Application

Muhammad Ali Gulzar, Chair

Miryung Kim

Daniel J. Williams

Eli Tilevich

Binoy Ravindran

June 9, 2026

Blacksburg, Virginia

Keywords: Automated Testing, Fuzzing, Scalable Computing

Copyright 2026, Ahmad Humayun

Automated Testing for Data-Intensive Scalable Computing

Ahmad Humayun

(ABSTRACT)

Data-Intensive Scalable Computing (DISC) systems such as Apache Spark, Hadoop, Flink, and Beam have become central to modern data processing. These systems enable developers to write applications as dataflows composed of user-defined functions operating over large and often unstructured datasets. However, testing such applications and the underlying frameworks remains a significant challenge. Conventional testing techniques struggle in this space due to the complexity of dataflow semantics, the scale and heterogeneity of input data, and the need to reason about operator interactions, schema variability, and program logic simultaneously. This thesis presents a set of techniques for extracting and leveraging rich, fine-grained properties that are unique to DISC workloads in order to inform automated input generation for effective testing across various levels of the data-centric software stack. The first thrust of this work focuses on generating inputs that can avoid trivial parsing errors and effectively exercise the deeper logic in DISC applications. By analyzing how code interacts with different parts of the input data, we test the code where it matters most instead of wasteful fuzzing cycles finding parsing issues. The second thrust explores how to create realistic and meaningful test data that reflects the structure and semantics of real-world inputs, while still achieving high coverage and fault detection. The final part of this work shifts focus to DISC frameworks, recognizing that applications are only as reliable as the systems they run on. By generating diverse dataflow programs, we systematically test internal components like optimizers, extending fuzzing to the entire DISC stack.

Automated Testing for Data-Intensive Scalable Computing

Ahmad Humayun

(GENERAL AUDIENCE ABSTRACT)

Modern data-intensive applications operate at extraordinary scale; historical sales analyses at companies like Amazon may require examining billions of records. To handle such volumes, practitioners rely on *Data-Intensive Scalable Computing* (DISC), a paradigm in which a program coordinates thousands of machines to process massive datasets in parallel. DISC frameworks, such as Apache Spark and Hadoop, provide the underlying execution machinery, allowing developers to express complex data transformations without managing the low-level details of distributed computing by hand. As DISC programs and their frameworks grow in complexity, automated testing techniques designed for conventional single-machine software are proving inadequate, creating an urgent need for new approaches tailored to the distributed setting. This thesis presents novel methods for automated testing across the full data processing stack. The first contribution proposes a technique for detecting faults in programs that combine and cross-reference multiple large datasets. The second develops methods that generate realistic-looking input data while simultaneously exposing program defects. The third introduces techniques to test the underlying infrastructure that coordinates and executes DISC computations, a layer that prior work has largely left untested.

Dedication

To my father, whose steady composure I inherit and call upon in turbulence.

To my mother, who would level every mountain to clear my path.

To my wife, with whom a lifetime isn't enough.

And to my siblings, who will always be my parents' second favorites :)

Acknowledgments

This work is funded by the National Science Foundation awards 2106420, 2452818, and 2541721. Amazon-Virginia Tech Initiative in Efficient and Robust Machine Learning, 4-VA, and the Commonwealth Cyber Initiative (an investment in the advancement of cyber R&D, innovation, and workforce development).

I am thankful to my committee, each of whom has contributed more to this journey than just being on the committee. Eli provided academic support to a close friend during a difficult period for him at VT. His advocacy for students as Associate Department Head makes a big difference to students' wellbeing. Dan's graduate OS courses taught me several deeper skills in research. He is one of those rare, gifted teachers who can break complicated topics down to their core essence. I'm glad he switched to academia; it will benefit generations of students to come. Three years ago, I took a graduate course on Compilers, masterfully taught by Binoy. The course ultimately played a pivotal role in securing my first position after completing my PhD. Miryung has been a close, senior collaborator and brilliant mentor throughout my degree. Despite being a juggernaut of a professor—one who is also Vice Chair of Graduate Studies at UCLA and has produced several other tenured professors—I never once felt like a burden on her time. She replied to my messages faster than most of my peers did, turned around every letter of recommendation without delay, and made me feel, consistently, as though I was her student rather than one of the many collaborators she has. Finally, and most importantly, I want to thank my advisor Gulzar. Through his mentorship, I have learned not only how to conduct rigorous research but also how to navigate the space between idealism and practicality, a lesson that has shaped both my academic and professional development. Even outside of direct guidance, he is someone I learned from

simply by observing him work. His work ethic, perseverance, and grit stand out, and it is genuinely inspiring to watch the dedication and effort he brings to everything he does. I hope to mirror these same qualities in my own work and mentorship throughout my career. I want to highlight his support especially during the final months of my degree, during which he actively helped me with the job hunt. He also gave me room to prepare for interviews while research progress slowed. Even as I saw other students struggle to find funding, it was one problem I never had to worry about. This alleviated a lot of stress that helped me focus. Having a competent and understanding advisor is a great blessing and I am grateful.

I am deeply grateful to the mentors and teachers who have shaped my journey through undergrad and beyond. Chief among them is my undergraduate advisor, Fareed Zaffar, whose tireless dedication ensures that hardworking students receive every opportunity they deserve. Junaid Haroon, an exceptional teacher whose advanced programming course I had the privilege of taking, fundamentally transformed the way I approach abstraction and problem-solving. Finally, Aman Goel, my mentor during my internship at Amazon, taught me the invaluable skill of decomposing research problems into their smallest constituents, and has continued to generously invest his time in my growth long after my time at Amazon. Much of my PhD was spent collaborating with Yaoxuan, one of the few people with whom collaboration feels genuinely effortless. I remember a particularly stressful submission where some of the results were not making sense and I was close to pulling out of the deadline entirely. In the final hours, he stepped in and debugged my experimental code while I kept my focus on the writing. That submission was NaturalFuzz, which was accepted and became the fourth chapter of this thesis. In Blacksburg, I was lucky to be surrounded by people who made both the work and the days themselves better. In the lab, Waris was a reliable thinking partner, always up for the kind of debates that helped clarify my ideas. Sabaat brought warmth and good conversation whenever I came in, and Tien was a steady presence throughout. Hadi went out of his way more times than I can count, keeping us overdosed

on his Biryani and always proactively offering to help. Before I got a car, he drove me to Washington, D.C. in the dead of night, to barely make a visa appointment for my first international conference. Housemates can be a genuine source of stress over the course of a PhD, so I consider myself fortunate that across five years, mine were uniformly good ones. Baig, Danish, Usama Saeed, and Faizan, each at different points along the way, collectively made the hard days easier simply by making the house feel like a place worth coming back to. Baig is the person I have shared a roof with the longest outside of family. He had a way of keeping the house alive, always present in the common areas, always ready for conversation. But more than that, I have learned a lot from simply living with him: to think a level deeper on things that matter, and to stay grounded, religiously, spiritually, and culturally. Usama Saeed was my first housemate in the US. We spent our first month effectively homeless, having missed our lease signing due to visa delays. We shared a living room lounge at other friends' house while navigating the relentless demands of building a life from scratch in a new country. Danish and Faizan consistently made food when times were tough. Faizan's chai with desi ghee parathay, something I hadn't had in years in the US, was a very nostalgic activity. Nameer, though our next door neighbor, was an honorary resident at our house. I will fondly remember the impromptu trips to the second-nearest Sheetz at 2AM, because the nearest one was a bit too close for a destressing trip. Nomair, Khizer, and Usama Munir have a gift for community and showing up for people, and the social fabric of my time in Blacksburg owes a great deal to both of them. Ibrahim and Maha, Ammar and Mahrukh, Haris and Sidrah, Adnan and Alifya: though they entered the story in its second half, they have made themselves a permanent, and significant part of it. There are many more in the Blacksburg community than I can name here, but each of them, in their own way, helped make what could have been a small town feel like home.

One of the quiet gifts of my PhD years was realizing that I had built a home not in one place, but across the country, and that wherever I went, there was someone to return to. Beyond

Blacksburg, old friends made sure I was never too far from familiar faces. Arham and Mariam have been an invaluable part of my life outside of work. During my early years in the US, few people were as attuned to my emotional state as they were. Their one-year-old daughter, Noor, deserves special recognition for offering therapeutic levels of cuteness through perhaps the most discouraging part of the PhD: the job search. Just a few miles away, Azka's company had a reliable way of restoring my energy when the PhD had worn it thin. Mahad has been a steady presence across years and distance, someone I've traveled with and always been able to talk to. I am deeply grateful to Farah, my wife's housemate, who always made space for me during my visits, helping make the challenges of long distance a little easier to bear. Not only did she graciously tolerate my extended stays, but she also indulged my requests for her signature khao suey. More broadly, the entire Pakistani student community in Madison made the US feel a little more like home. Their friendship, generosity, and sense of community provided comfort and belonging throughout my time there. Furqan, Ayaz, Punnaal, Abdullah, Suri, Jojo, and Mehreen have been my friends since before graduate school. Some of my fondest memories from these years come from the trips we took together, exploring different corners of the US and creating memories that I will cherish for years to come.

I want to thank my parents and siblings for the constant support throughout this endeavor. At the end of any day, I always had someone to share my day with. To keep updated. People who celebrated every victory, consoled me through every loss. My mother never once compromised on my education. In drafting this acknowledgement, I wrote several times that she "moved mountains" to put me in the best schools we could afford, but that phrase fails to capture the fierceness and urgency with which she fought, sacrificed, and refused to accept anything less than the best she could give me. Her advocacy was relentless. I write this knowing that so much of what she has done has gone unspoken and unacknowledged, even by me, who was too young to really understand what she was doing for me. And so I enshrine it here: her quiet, unwavering sacrifice is woven into every page of this work,

and into the person I have become. My father, apart from being the primary breadwinner and playing his part in everything my mother did for my education, has been a source of wisdom whenever I have needed it. Back when I was about to buy my first car, something I needed but carried great financial risk as a PhD student, I looked to him for advice. Rather than injecting his own opinion, he taught me the mindset needed to commit to financially consequential decisions. That mindset has endured, surfacing every time I have faced an important decision. My elder brother has always been someone I looked up to, and for much of my life, I followed his academic path by instinct. Being only a year and a half older, he had a way of arriving at every frontier just ahead of me, absorbing the uncertainty so that my own way forward always felt a little clearer. I entered undergrad fully convinced I would study Electrical Engineering like him, before eventually finding my own direction. Among a lineup of brothers, my younger sister is the emotional core of our family. Her gifts, some bought with careful selection and others crocheted by hand, have quietly accumulated across every corner of my life, on my desk, in my room, in my car. They serve as small reminders that I have loved ones even when I am far from home. One of her trinkets was a small astronaut perched on the moon, which I placed on top of a large wall hanging in my room. It was a precarious spot, just unstable enough that opening the bedroom door too quickly would send it tumbling. I realize now that every time it fell, it would inevitably remind me of home when I went to place it back. My curious little brother, barely in high school, would often ask me what I was working on. Having to explain my research to him pushed me to look at it from different angles, find the right analogies, and strip it down to what actually mattered. It turns out that is not so different from what good science communication requires.

Finally, I want to thank my wife, Rabeeya. Even though we were long-distance, her companionship made the journey easier. While my family knew my life in broader strokes, Rabeeya was the one I debriefed with on the day-to-day. Pursuing a doctorate herself, she had a visceral understanding of my daily challenges. We are, in many ways, two parts of the same

soul, and having her in my life has made it much more full and fulfilling.

Contents

List of Figures	xv
------------------------	-----------

List of Tables	xviii
-----------------------	--------------

1 Introduction	1
1.1 Background and Motivation	1
1.2 Contributions	5
1.2.1 Dependency Guided Input Generation for DISC Programs	5
1.2.2 Natural Input Generation via Interleaving Mutations	6
1.2.3 DAG-based Workload Generation for DISC Frameworks	7
2 Review of Literature	8
2.1 DISC Application Testing	8
2.2 Taint-Based Fuzzing	9
2.3 Fuzzing Rare Paths	9
2.4 Provenance Techniques in DISC	10
2.5 Data Generation for Database Testing	11
2.6 Semantic and Grammar-Based Testing	11
2.7 SQL Generation and Fuzzing	12

2.8	API Fuzzing	13
2.9	LLM-Based Fuzzing	13
3	Co-Dependence Aware Fuzzing for Dataflow-based Big Data Analytics	15
3.1	Introduction	15
3.2	Motivating Example	18
3.3	Approach	21
3.3.1	Phase I: Enabling Fine-Grained Taint Analysis	25
3.3.2	Phase II: Fine-Grained Taint Tracking	26
3.3.3	Phase III: Row Selection for Data Size Reduction	28
3.3.4	Phase IV: Co-Dependence Aware Mutation	29
3.4	Evaluation	30
3.4.1	Test Coverage Against Baseline	33
3.4.2	Fault Detection	36
3.4.3	DEPFUZZ’s Instrumentation Overhead	42
3.4.4	Limitations of Dynamic Taint Analysis	43
3.5	Chapter Summary	45
4	Natural Input Generation for Big Data Analytics	46
4.1	Introduction	46
4.2	Motivating Example	49

4.3	Approach	53
4.3.1	Interleaving Input Mutation	54
4.3.2	Fine-grained Branch Profiling	55
4.3.3	Computing Path Vectors	57
4.3.4	Fuzzing with Interleaving Mutation	58
4.4	Evaluation	61
4.4.1	Code Coverage	64
4.4.2	Error Detection	66
4.4.3	Synthetic Data Generators for Testing	67
4.4.4	Test Input Naturalness	70
4.4.5	Threats to Validity	72
4.5	Chapter Summary	73
4.5.1	Future Outlook	73
5	DAG-based Workload Generation for Testing Data Intensive Scalable Computing Frameworks	74
5.1	Introduction	74
5.2	Motivating Example	78
5.3	Methodology	80
5.3.1	Stage 1: DAG Generation	81
5.3.2	Stage 2: Abstract DFG Construction	83

5.3.3	Stage 3: State-Aware DFG Parameterization	85
5.3.4	Code Generation for Target Engines	88
5.3.5	Oracles for Bug Detection	89
5.4	Evaluation	90
5.4.1	Optimizer Reachability Rate (RQ1)	92
5.4.2	Optimizer Coverage Analysis (RQ2)	94
5.4.3	Bugs and Semantic Depth (RQ3)	99
5.4.4	Generalizability (RQ4)	103
5.5	Chapter Summary	104
6	Future Directions	105
6.1	Testing Streaming Applications	105
6.2	Rule-Targeted Testing	106
6.3	Cross-Framework Differential Testing	107
7	Conclusion	109
	Bibliography	111

List of Figures

3.1	A DISC application with a fault in the UDF of step 9 , map : (a) code in Scala. (b) the corresponding dataflow graph, and (c) an illustration of data manipulation in 9 steps. Blue and red colored texts are co-dependent regions identified by DEPFUZZ	18
3.2	A sample configuration file required by BigFuzz . The file contains schemas and seed inputs for the flights and airports dataset, and a user-specified time cut-off for the fuzzing campaign.	20
3.3	A test case generated by DEPFUZZ that causes a NaN exception in the program in Figure 3.1	20
3.4	Workflow of DEPFUZZ	21
3.5	The red color rows do not participate in the join operation since the filter operation removed them earlier.	22
3.6	Taint analysis enabled String type in Scala	24
3.7	Taint propagation through a simple dataflow program. Yellow colored highlighted text is the provenance of red colored text at the bottom left table. . .	25
3.8	Statement coverage of three tools on 17 benchmark programs during 24 hours	34
3.9	Cumulative number of faults detected during 24 hours averaged across 17 programs. (b) shows the average for P5-P17 which ingest a single dataset, and (c) shows average for P1-P4 which take multiple datasets as input.	37

3.10	Depth vs Time for all faults detected by DEPFUZZ and baselines. $\otimes$ denotes centroids of each tool. DEPFUZZ has more points near the top left corner, which means it detects <u>deeper</u> faults <u>faster</u> than baselines. DEPFUZZ finds <u>more</u> faults than baselines.	38
4.1	Dataflow graph of a program that computes the total sales price of items from a particular manufacturer for each year. The <code>+</code> operator in the <code>reduceByKey</code> should be a <code>-</code> and is incorrect, leading to incorrect values in the output. . .	49
4.2	Sample inputs generated by each tool.	50
4.3	How interleaving unlocks new paths in the program	53
4.4	Branch predicate expressions, bit vectors, and augmented datasets constructed by NATURALFUZZ	54
4.5	Taint analysis enabled <code>Int</code> type in Scala	56
4.6	Statement coverage progress of NATURALFUZZ and baselines on eight benchmark programs.	61
4.7	Tainted String on line 7 is casted to String on line 8, resulting in a loss of taint. This can cause NATURALFUZZ 's expression capture to be incomplete. .	66
5.1	Dataflow programs in various frameworks exhibit a similar, DAG-based sub-structure.	75
5.2	Walk through of DAGGER identified UDF non-determinism bug in Apache Spark. .	78
5.3	Four stage workflow of DAGGER	80
5.4	Format for the API specification.	82

5.5	Error distribution across all frameworks.	92
5.6	Line Coverage vs Time graphs for five trials of 24-hour fuzzing campaigns. The colored bands show one standard deviation, and lighter shades denote the min and max.	94
5.7	Branch Coverage for five trials of 24-hour fuzzing campaigns on Spark and PyFlink.	96
5.8	Examples of bug-inducing workloads generated by DAGGER . These programs expand into multi-stage execution DAGs involving repeated grouping, re- aggregation, operator reordering, and UDF boundaries. Although minimized for clarity, they induce deep planner pipelines with non-trivial operator depen- dencies that cannot be expressed as a single declarative SQL query, placing them outside the reach of SQL-only fuzzers such as SQLSmith.	100

List of Tables

3.1	Summary of how each class of operators produces co-dependent regions in the input dataset. For simplicity, we use <code>row[1].col[3]</code> as a human-readable representation of input byte region $b_i, \dots b_j$, where $0 < i < j < size(dataset)$	27
3.2	Subject programs used in DEPFUZZ 's evaluation. All programs represent real-world DISC use cases and are adopted from prior work. The data and code characteristics of benchmark programs are also shown. D , O , d , and R denote the number of datasets, number of operators, maximum operator depth, and total rows, respectively.	31
3.3	Running time of the original subject program and the instrumented program with taint analysis along with total errors detected by each tool.	41
3.4	Theoretical limitations of implicit taint propagation. In each example, t is a tainted primitive type. Occurrences denotes the number of programs in which the pattern appears.	43
4.1	Benchmark programs and their descriptions as taken from the TPC-DS Specification [7]	62
4.2	Average overhead and average total errors detected by each tool.	64
4.3	Comparison of coverage, fault detection, and generation time between SDV and NATURALFUZZ . Time is reported in minutes.	69
4.4	Average naturalness scores of inputs generated by NATURALFUZZ and baselines.	71

5.1	Top component-level percentage line coverage differences between DAGGER and SQLSmith for each framework.	97
5.2	Previously unknown bugs in DISC frameworks discovered by DAGGER in 24-hour test generation campaigns.	101
5.3	Lines of code comparison across frameworks for Dagger vs SQLsmith. codegen covers operator and query generation logic (C++ for SQLSmith, Scala for DAGger). environment covers runtime configuration, test launching, and oracle implementation; SQLSmith requires additional binding code since it lacks native DISC framework support.	103

Chapter 1

Introduction

1.1 Background and Motivation

Modern large-scale computing systems increasingly rely on dataflow-oriented programming models to express and execute complex workloads. Frameworks in the Data-Intensive Scalable Computing (DISC) domain, such as Apache Spark and Hadoop MapReduce, offer high-level APIs that allow developers to compose programs as Directed Acyclic Graphs (DAGs) of operators like map, join, and groupBy, often embedding user-defined functions (UDFs) within these operator chains [51]. This abstraction enables efficient parallelism and composability across distributed clusters, making it a common foundation not only for big data analytics but also for machine learning systems like TensorFlow [11], which use dataflow graphs to represent both computation and mutable state.

However, despite their popularity, these frameworks and the programs that run on top of them, remain difficult to test in a systematic way. Their execution semantics involve deep inter-operator dependencies, large-scale input datasets, and UDFs with potentially non-deterministic behavior. These characteristics make bugs hard to detect and reproduce, especially semantic faults that arise only under specific configurations or optimization rewrites. Existing automated testing methods are ill-suited for such settings due to their inability to reason about the structural and semantic complexity of dataflow-based computation [101].

Automated testing for DISC applications, and their underlying frameworks, face three key challenges:

- DISC applications operate over massive, structured input spaces governed by implicit constraints (*e.g.* join keys across datasets) that are difficult to satisfy systematically. This makes it hard for any testing approach to achieve meaningful code coverage or surface deep bugs.
- DISC applications are designed to process natural, production-like data, yet inputs that reflect real-world distributions rarely drive applications through diverse execution paths. Reconciling input realism with broad path coverage is a fundamental tension in testing DISC systems.
- DISC frameworks expose their core functionality through dataflow APIs that no general-purpose testing technique currently targets directly. SQL-oriented approaches can partially substitute when a framework offers an SQL frontend, but they cannot express stateful operators and UDF-heavy pipelines, and offer no foothold at all on frameworks without SQL support.

Because of the unique challenges in the DISC domain, existing automated test generation techniques are often ineffective. This thesis aims to close that gap.

Fuzzing is an effective software testing approach for many complex programs [2, 21, 25, 43, 64, 69, 85, 102, 111]. Fuzzers make small perturbations (mutations) to inputs to increase the likelihood of exercising uncovered application logic. Such traditional fuzzing may take a long time to generate meaningful inputs for DISC applications because a large input data has *too many locations* to mutate. Not only is it necessary to identify which rows and columns are worthwhile to mutate, the generated input must also satisfy complex constraints from

mixing dataflow operators and user-defined functions. Mutations used in fuzz testing today fail to account for such *co-dependence* and thus may not exercise application logic beyond certain dataflow operators. This problem is further exacerbated because, unlike SQL, there is no explicitly defined schema to identify columns, and the inputs for DISC applications are usually parsed on the fly. This thesis investigates how to analyze the interaction between programs and their input data to uncover co-dependencies across different parts of the data, and how to leverage this information to guide more effective and targeted input mutations during fuzzing.

Moreover, even when DISC fuzzers achieve high code coverage, they tend to produce unnatural and unreadable test inputs that hinder debugging and developer trust. This is because most fuzzers are designed with the primary goal of maximizing code coverage, without considering the naturalness or realism of the generated inputs as an objective. While machine learning-based synthetic data generators like Synthetic Data Vault (SDV) [84] can create realistic-looking datasets by learning patterns from existing data, they are intended for use cases such as simulation or privacy-preserving analytics. These tools are not designed to explore diverse program behaviors or execution paths, and therefore fall short when applied to software testing. As a result, they perform poorly when it comes to exploring diverse execution paths or triggering deep semantic logic within programs, leading to low code coverage. Additionally, such generators typically rely on well-defined schemas or explicit type information to learn and generate new data. This requirement limits their applicability in big data analytics contexts, where input datasets are often unstructured or semi-structured and field types are inferred dynamically at runtime rather than declared upfront. This thesis introduces techniques that leverage the abundance of data in typical DISC deployments to generate unseen data via *interleaving* mutations, striking a balance between exploration and naturalness.

While generating high-quality input data is essential for exercising application logic, testing the underlying DISC frameworks themselves requires generating entire programs, specifically, diverse and valid compositions of dataflow operators that reflect real execution patterns. DISC frameworks like Apache Spark and Hadoop MapReduce offer high-level APIs that allow developers to construct computation pipelines as Directed Acyclic Graphs (DAGs) of operators such as `map`, `join`, and `groupBy`, with user-defined functions (UDFs) embedded within these flows [51]. This abstraction promotes modularity and parallelism, making it foundational not only for big data processing but also for modern machine learning systems like TensorFlow [11], which represent both computations and mutable state as dataflow graphs. Figure 5.1 illustrates code examples across different dataflow APIs. Despite their popularity, these frameworks remain difficult to test systematically. The combination of deeply composed operator chains, large-scale data movement, and often non-deterministic behavior of UDFs leads to subtle and hard-to-detect bugs [101]. Traditional testing techniques are ill-equipped to handle these complexities, especially when bugs do not manifest as crashes but rather as silent semantic errors. To date, there is no general-purpose fuzzing framework that targets dataflow APIs in a principled way. While automated test generation has proven useful in specialized domains such as deep learning API testing [104], database fuzzing [18, 61, 93], and DISC application-level testing [51, 53, 105, 111], these approaches are either limited to specific domains or tightly coupled to particular input formats (e.g., SQL). What is still missing is a framework-agnostic fuzzing approach capable of synthesizing arbitrary operator DAGs, including complex compositions of joins, filters, unions, and UDFs, to comprehensively exercise the execution engines and optimizers that operate beneath high-level dataflow interfaces. This thesis addresses this gap by proposing the first DAG-based fuzzing framework for dataflow-oriented APIs by treating the DAG of operations as a first-class artifact for test generation.

The central hypothesis of this thesis is that we can develop a set of techniques for extracting and leveraging rich, fine-grained properties that are unique to DISC workloads in order to inform automated input generation for effective testing across various levels of the data-centric software stack. We explain our contributions and the planned future work in detail in the following section.

1.2 Contributions

This thesis presents three contributions that build on the central idea of analyzing and leveraging fine-grained workload structure to improve testing of DISC systems.

1.2.1 Dependency Guided Input Generation for DISC Programs

The first contribution is a technique for dependency-guided test input generation at the DISC application layer. The goal is to generate test data that exercises hard-to-reach regions of application code by understanding how data is used across the program. The approach performs fine-grained taint tracking to establish which rows and columns of which datasets influence specific parts of the application logic. It further identifies co-dependence constraints across datasets induced by dataflow operators such as `join`, enabling coordinated mutations across datasets to satisfy these constraints.

To realize this approach, Spark programs written in Scala are instrumented by overriding dataflow operators and UDFs to capture row-level, column-level, and dataset-level provenance. Using this provenance, fuzzing retains only the relevant parts of input data for mutation, significantly improving efficiency without sacrificing code coverage. Experimental evaluation on 17 DISC applications shows that the technique outperforms prior fuzzers like Jazzer and

BigFuzz in coverage, fault detection, and depth of faults uncovered.

We instantiate this idea in a tool called **DEPFUZZ**, which is publicly available, dockerized¹, and runnable [52].

1.2.2 Natural Input Generation via Interleaving Mutations

The second contribution is a technique for natural input generation via interleaving mutations that extends dependency-guided provenance but shifts focus toward test input quality and realism. The goal is to maintain the naturalness of test data while still achieving high coverage and fault detection. The approach profiles program execution to understand which parts of the input influence branching behavior, then applies interleaving mutations that splice together meaningful parts of different rows and columns.

This method leverages data provenance to map branching conditions to input fields, allowing it to generate realistic synthetic inputs that retain semantic consistency. It is evaluated on TPC-DS-based applications and achieves significantly higher coverage and fault detection than prior fuzzers, while also producing inputs that are demonstrably more natural based on perplexity metrics. The generated outputs are more readable and closer to real-world data, which aids debugging and makes the approach more appealing to developers.

We develop a prototype implementation of this technique called **NATURALFUZZ**, which is available on github as a dockerized and runnable tool².

¹<https://github.com/SEED-VT/DepFuzz>

²<https://github.com/SEED-VT/NaturalFuzz>

1.2.3 DAG-based Workload Generation for DISC Frameworks

The final contribution is a technique for DAG-based workload generation that broadens the scope of fuzzing to the DISC framework layer. While the previous two contributions focused on fuzzing applications written using dataflow APIs, this approach targets the frameworks themselves, especially their query planning and optimization components. The idea is to synthesize logical operator DAGs as first-class test artifacts rather than relying on surface-level grammars or SQL strings.

The approach incrementally lowers logical DAGs into semantically valid executable programs across Apache Spark, PyFlink, Polars, and Dask, enabling fine control over operator composition, UDF placement, and data dependencies. This enables the discovery of optimizer-level faults and semantic inconsistencies that traditional SQL-based fuzzers cannot expose. A plan-based oracle detects violations of non-deterministic UDF contracts, and the technique has uncovered multiple previously unknown bugs in production DISC optimizers. Compared to SQLSmith, it achieves substantially higher optimizer reachability and coverage.

Together, these three contributions demonstrate how extracting structured workload knowledge and building targeted mutation strategies can systematically improve the quality and scope of automated testing across the entire DISC stack.

We instantiate this idea in a tool called **DAGGER**, which is open-source and has discovered several faults in real-world DISC frameworks³.

³<https://github.com/ahmayun/dag-fuzzer-generalized/tree/main>

Chapter 2

Review of Literature

2.1 DISC Application Testing

Prior work has explored a range of challenges in testing DISC applications. **BigFuzz** [111] is a black-box fuzzing technique that uses schema-aware mutations to mitigate trivial parsing errors, increase coverage, and employs framework abstraction to allow local fuzzing of DISC applications. Symbolic execution techniques [48, 66, 67, 81] also exist for testing DISC applications; among these, **BigTest** [48] was the first to explore framework abstraction for this purpose. **NaturalSym** [105], augments typical symbolic execution to bias generation towards more natural looking inputs. However, symbolic execution approaches cannot easily generate constraints that respect co-dependence relationships within multiple datasets, created by the complex interaction between dataflow operators and UDFs.

Fuzzing has gained popularity in industry and academia relatively recently due to its black-box nature and ease of adoption [82]. A common challenge is generating structurally valid inputs; **Zest** [83] attempts to address this using parametric generators. While there is no work that specifically targets generating dataflow programs, several tools have built data fuzzers for DISC programs. Tools like **Titian** [58] and **BigDebug** [46] focus on data provenance and interactive debugging, providing insights into the lineage of data and facilitating identification of root causes in complex dataflows. These works target finding faults in top-level DISC applications rather than the frameworks themselves. This thesis presents a set of techniques

for holistic testing of the entire stack i.e. the application and also the framework.

2.2 Taint-Based Fuzzing

One of the closest lines of work to some tools developed in this thesis is taint-based fuzzing. At a high level, all taint analysis techniques attempt to isolate regions within an input that are critical to mutate. For example, Bekrar *et. al.* [21] propose taint-based fuzzing that identifies input regions on which to focus mutations. Other security-focused techniques such as TaintScope [102] and BuzzFuzz [43] attempt to isolate regions of the input used inside sensitive library calls to increase the chances of finding security bugs. PATA [69] performs path-aware taint analysis to mitigate over-tainting and under-tainting by employing path information. Although these techniques isolate critical input regions, none target DISC applications, and none can discover underlying co-dependence relations by analyzing dataflow operators and UDFs. The inputs to DISC applications are very large and consist of multiple datasets, making existing byte-level taint tracking inefficient. Work in this thesis addresses these problems by handling multiple datasets and tracking taints at the level of dataset IDs, columns, and rows from unstructured inputs. Additionally, none of these fuzzers target realistic input generation; work in this thesis on natural fuzzing for DISC applications is the first to incorporate the notion of naturalness in this context.

2.3 Fuzzing Rare Paths

The idea of triggering *hard-to-reach* regions of a program has been explored frequently in the literature. FairFuzz [64] is a targeted mutation strategy that avoids mutating input regions triggering rare branches; work in this thesis similarly analyzes the use of fusion operators to

co-mutate certain regions, though it uses dataflow structure rather than a simple masking strategy. AFLFast [25] prioritizes inputs that trigger rare *paths* in the code by instrumenting the program binary and performing runtime coverage analysis. Both FairFuzz and AFLFast are unsuitable for DISC applications because they do not analyze dataflow operator usages and internal UDF semantics to infer co-dependent input regions in large datasets, and neither performs provenance-aware duplication to resolve aggregations, which are extremely common in DISC applications.

Driller [99] switches to symbolic execution to resolve difficult branches that AFL fails to pass, inheriting the limitations of symbolic execution. Steelix [68] attempts to produce a single input passing a difficult-to-hit branch and employs source-level instrumentation similar to some tools in this thesis; however, it is unsuitable for DISC applications with large inputs due to a lack of fine-grained data tracking.

2.4 Provenance Techniques in DISC

TaintStream [109] implements cell-level provenance for Apache Spark in the context of policy enforcement. Some tools developed in this thesis also track provenance at the cell level, but unlike TaintStream—which requires extending the original dataset with tags—their cell-level tracking is fully automatic and requires no conversion of the original dataset. The taint analysis used in this thesis is similar to that of FlowDebug [100], as both instrument primitive data types and application code without requiring modifications to original datasets. However, FlowDebug concerns taint analysis only and does not generate test data nor identify co-dependency constraints among input datasets.

Existing data provenance techniques [29, 54, 55, 56, 72] perform taint analysis only at the row level, support only a single dataset, and do not support tracking at the column or cell

level. Spark-specific solutions such as Titian [58] are similarly limited to row-level provenance for a single input dataset. BigSift [47] extends delta debugging for DISC applications but isolates at the level of rows rather than at the level of dataset IDs, rows, and columns, as work in this thesis does.

2.5 Data Generation for Database Testing

Parts of this thesis have parallels with database testing [12, 24, 50]. Houkjær *et. al.* [50] propose a graph-based method to guide data generation. Bruno and Chaudhuri [24] propose a language-based technique for general-purpose data generation. Other approaches target schema coverage [12]. These tools generate synthetic data with the goal of testing database performance and workloads, and none target DISC applications. In contrast, work in this thesis on natural fuzzing utilizes real-world data to generate minimal and realistic test inputs with the goal of discovering program faults.

We also compare against ML-based fake data generators such as SDV [84] and more primitive synthetic generators such as Faker [8], which have been used to generate fake data in prior work [109]. However, these synthetic data generators require manual configuration and do not account for the complex constraints introduced by dataflow operators in the program.

2.6 Semantic and Grammar-Based Testing

Another related line of work is automated testing to find semantic faults. Zest [83] relies on program feedback to find semantic faults while avoiding syntax errors. Other works rely on context-free grammars of the input domain to generate inputs that avoid syntax errors and better explore the semantic stages of the program under test [22, 30, 44, 75, 97]. However,

these techniques require grammar rules that are complex to write or infer. In contrast, the natural fuzzing approach developed in this thesis does not rely on any grammar rules.

2.7 SQL Generation and Fuzzing

SQLsmith [93] uses a grammar-based approach to generate syntactically valid SQL queries, prioritizing always-valid syntax at the cost of query complexity. SQLancer [18] generates semantically valid SQL queries using a rule-based approach and detects logic bugs via oracles like Pivoted Query Synthesis and Ternary Logic Partitioning. CERT [19] extends SQLancer by identifying issues in cardinality estimation by the optimizer, which can cause sub-optimal query plans. Squirrel [112] improves upon grammar-based approaches by using coverage-guided fuzzing and type-aware mutations over an intermediate representation (IR) of SQL. DynSQL [61] advances SQL fuzzing further by interleaving query generation with execution, dynamically extracting the updated schema and data state after each statement.

Tools such as Thanos [42] and Puppy [103] target the storage system or plan generator of specific DBMSs, restricting their testing to one-off campaigns against a single system. Work in this thesis, by contrast, operates at the dataflow level rather than SQL syntax, enabling fuzzing of a broader class of applications and transformations that are difficult to reach with SQL-specific generation alone. Unlike DynSQL, the dataflow-level tool developed in this thesis does not need to execute a query to identify state updates; it instead uses the dataflow graph and API specification information to perform high-level updates to state.

2.8 API Fuzzing

API fuzzing has evolved to handle stateful interfaces where call sequences and dependencies are critical. RESTler [17] analyzes OpenAPI specifications to generate dependency-respecting request sequences; MINER [73] combines data-driven techniques with machine learning to predict critical parameters; and APIRL [41] uses deep reinforcement learning to adaptively explore API behaviors. However, these tools focus predominantly on RESTful services and require existing API specifications. Work in this thesis operates at the dataflow level, generating and testing complex operator DAGs without requiring detailed API schemas, offering a schema-agnostic approach applicable to diverse data processing frameworks.

2.9 LLM-Based Fuzzing

Large language models (LLMs) have recently been applied to fuzzing, leveraging their ability to generate syntactically valid and semantically meaningful inputs. Fuzz4All [106] is a universal fuzzer that uses LLMs to generate and mutate inputs across a variety of programming languages and input formats, relying on an “autoprompting” strategy to distill a system prompt from user-provided specifications. TitanFuzz [35] and FuzzGPT [36] apply LLM-based generation to deep learning library APIs, using models to synthesize programs that trigger edge-case behavior in frameworks like TensorFlow and PyTorch. CodaMosa [65] integrates LLMs into search-based test generation to escape coverage plateaus, querying a code model when traditional search makes no progress. OSS-Fuzz-Gen [70] uses LLMs to automatically generate fuzz harnesses for open-source projects, reducing the manual effort typically required to bootstrap fuzzing campaigns. ChatAFL [77] leverages LLMs to enrich protocol mutation in network fuzzing by inferring message structure from protocol specifications.

These techniques predominantly target single-language or single-framework settings and do not handle the multi-dataset input structure characteristic of DISC programs. Additionally, even when the technique is an end-to-end LLM-based general-purpose fuzzer (like Fuzz4All [106]), each iteration requires expensive LLM calls that require access to specialized hardware. Work in this thesis, by contrast, presents techniques where each fuzzing iteration is much cheaper than an LLM call and does not require any specialized hardware.

Chapter 3

Co-Dependence Aware Fuzzing for Dataflow-based Big Data Analytics

3.1 Introduction

Data-Intensive Scalable Computing (DISC) applications have become a prevalent way to process large-scale data. DISC frameworks like Hadoop MapReduce [3] and Apache Spark [4] offer APIs that contain dataflow operators such as `map`, `join`, and `groupByKey` for parallel data processing across thousands of machines. A typical DISC application builds on a series of dataflow operators in conjunction with user-defined functions (UDFs) that are passed as arguments to the dataflow operators. Despite the widespread usage of DISC applications, testing remains difficult due to their large input size and the applications' complex interactions with data.

Fuzzing is an effective software testing approach for many complex programs [2, 21, 25, 43, 64, 69, 85, 102, 111]. Fuzzers make small perturbations (mutations) to inputs to increase the likelihood of exercising uncovered application logic. Such traditional fuzzing may take a long time to generate meaningful inputs for DISC applications because a large input data has *too many locations* to mutate. Therefore, it is necessary to identify which rows and columns are worthwhile to mutate when a fuzzer attempts to reach a new code location. Naive mutations cannot satisfy complex input constraints from mixing dataflow operators and user-defined

functions. For instance, `join` concatenates rows from two datasets that have matching values in designated *key* columns. This introduces an implicit equality constraint between the fields of multiple datasets. Consequently, to exercise code inside the UDF `func1` of `map` in the code snippet `dataset1.join(dataset2).map(func1)`, input mutations must simultaneously operate on both datasets `dataset1` and `dataset2` to observe the co-dependence constraint *i.e.*, there must exist a row in `dataset1` with the same key as the `dataset2`'s first column in order for `join` to produce any data on which `map` can apply `func1`. Mutations used in fuzz testing today fail to account for such *co-dependence* and thus may not exercise application logic beyond `join`. This problem is further exacerbated because, unlike SQL, there is no explicitly defined schema to identify columns, and the inputs for DISC applications are usually parsed on the fly.

We propose **DEPFUZZ**, a fuzzer that performs *co-dependence aware* row selection and column mutation while ensuring that constraints among *multiple datasets* are observed. **DEPFUZZ** combines row-level and column-level data tracking via taint analysis. In other words, it identifies which rows and which columns from which dataset are operated by individual lines of application code. This knowledge of row-level provenance helps reduce data size for subsequent fuzzing iterations, as **DEPFUZZ** retains only selected rows and mutates them, as opposed to the entire dataset during mutational fuzzing without sacrificing code coverage. By inferring *co-dependence* relations among different columns from multiple datasets, it increases the chance of generating meaningful unstructured inputs that can reach the later stages of the application after operations such as `join` and `co-group` are used.

DEPFUZZ instruments the program under test by overriding dataflow operators and UDF components to capture row-level, column-level, and dataset-level provenance. This is done by implementing dynamic taint tracking for UDFs and dataflow operators. By leveraging co-dependence aware row selection and column mutations, it generates inputs that can reach

deeper regions (*i.e.*, UDFs in the later stages of dataflow operators).

To evaluate **DEPFUZZ**, we use 17 DISC applications and measure (1) statement coverage, (2) fault detection capability, and (3) fuzzing speed-up. To assess fault detection capability, we inject faults at different depths in terms of the program’s joint dataflow and control flow graph. We evaluate **DEPFUZZ** against two baseline techniques: Jazzer [57], a coverage-guided greybox fuzzer for Java bytecode based on LibFuzzer [94]; and **BIGFUZZ** [111], a greybox fuzzer for DISC applications. Comparison against **JAZZER** and **BIGFUZZ** serves to assess the overall benefit in terms of fault detection and speed-up, when orchestrating input mutations across multiple datasets by identifying co-dependence constraints. **DEPFUZZ** achieves 87% statement coverage, which is 29% and 13% more than **JAZZER** and **BIGFUZZ**. It also obtains coverage $2.1\times$ and $1.3\times$ faster than **JAZZER** and **BIGFUZZ**, respectively. Since faults appearing in earlier stages tend to be easier to find (*e.g.*, due to ill-formatted inputs) than those faults appearing in later processing stages, we evenly distribute injected faults in all dataflow operators for fairness. The average depth of a fault found by **DEPFUZZ** is 3.7 operators deep compared to 2.8 and 2.6 by **JAZZER** and **BIGFUZZ**, respectively. Our contributions are as follows:

- We present a new fuzz testing approach that leverages rich provenance information to increase mutational fuzzing’s effectiveness and efficiency for DISC applications. This is the first test generation approach that extracts co-dependence constraints at the level of rows, columns, and datasets *fully automatically* without requiring an explicit schema from a user.
- Our evaluation includes an extensive comparison against two baselines on 17 different benchmark programs for 24 hours each. The results show **DEPFUZZ** reaches previously uncovered code faster, finds faults faster, and reaches deeper code locations of later stages than existing fuzzers.

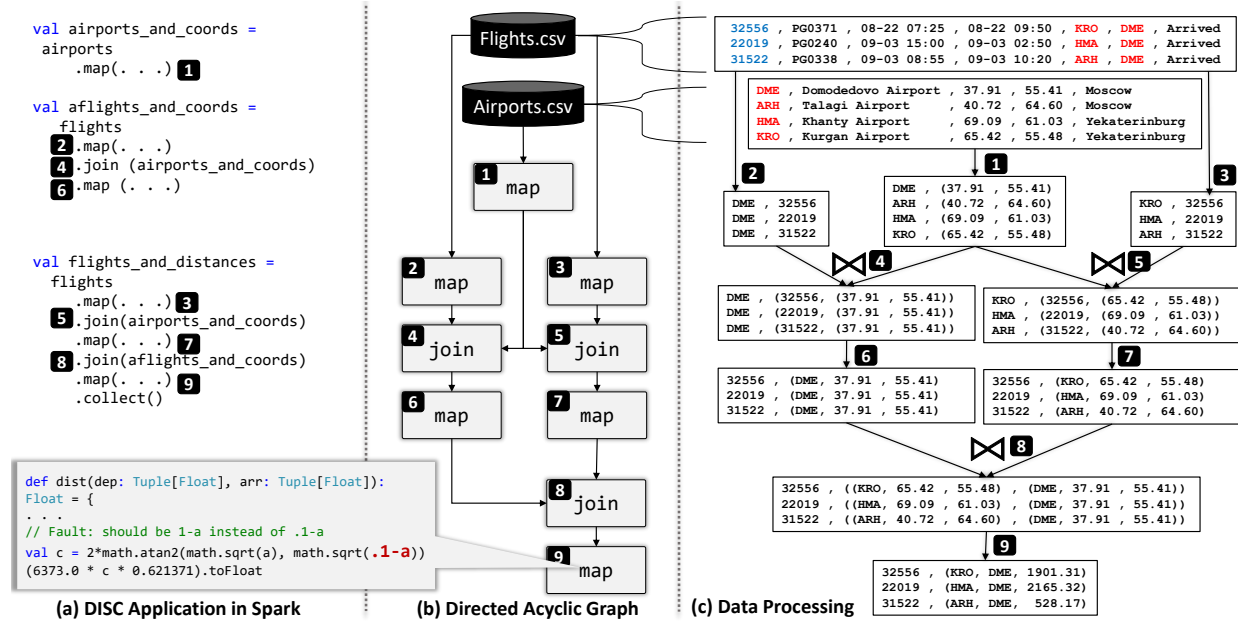


Figure 3.1: A DISC application with a fault in the UDF of step 9, map: (a) code in Scala. (b) the corresponding dataflow graph, and (c) an illustration of data manipulation in 9 steps. Blue and red colored texts are co-dependent regions identified by **DEPFUZZ**.

- **DEPFUZZ** is built on extended dynamic taint tracking and analysis of dataflow operators.

It has comprehensive support for Apache Spark-based DISC applications written in Scala, and its key idea generalizes to other dataflow-based big data applications such as Google’s MapReduce or Apache Hadoop. **DEPFUZZ** is publicly available at <https://github.com/SEED-VT/DepFuzz> [52]

3.2 Motivating Example

This section motivates **DEPFUZZ** with a concrete example. Suppose a data analyst computes the distance traveled by airplanes for each flight in 2017 from two input datasets: **flights** contains millions of flights flown worldwide in 2017 and **airports** contains the geographic location of airports. The top two boxes in Figure 3.1 (c) show sample rows from each

dataset. The `flights` dataset has a flight ID, the departure and landing times, the departure and arrival airport codes, and the flight status, all separated by commas. The `airport` dataset maps airport codes to their airport name, longitude and latitude coordinates, and the corresponding city. Figure 3.1 (a) shows a DISC application written in Spark. It consists of dataflow operators, such as `map` and `join`, where some dataflow operators, such as `map`, take a user-defined function (UDF) as an argument. For example, the `9` `map` takes a UDF that computes distance using the Haversine formula, shown in the expanded text box.

In `1`, the analyst extracts the airport code and longitude and latitude values from `airport`. From `flights`, she selects the departure and arrival airport codes and the flight ID, as the first column and the second column in `2` and `3`, respectively. She uses `join` in `4` and `5` to join the arrival and departure airports with their longitude and latitude coordinates. `8` joins the two data streams using a flight ID. `9` applies the `dist` function on the pairs of latitude-longitude tuples to compute the Haversine distance.

While writing the Haversine formula, she mistakenly writes `sqrt(.1-a)` instead of `sqrt(1-a)` (text box for `9` in Figure 3.1 (a)). This error is hard-to-spot and subtle and causes NaN exceptions.

Limitations of Existing Fuzzers. To reveal such errors, suppose that she runs a commercially used, coverage-guided greybox fuzzer, JAZZER [57]. After a 24-hour fuzzing campaign, even with coverage guidance, Jazzer cannot produce an input to reach code beyond custom parsing logic at `2`, where it persistently triggers the same `ArrayOutOfBoundsException`. JAZZER achieves a maximum statement coverage of 27%. Due to a lack of schema and a lack of awareness of co-dependent regions, it continues to generate random strings for the two datasets that cannot pass beyond the parsing stage (*i.e.*, `map` at `2`).

Similarly, BigFuzz [111] requires an input schema (as shown in Figure 3.2) to apply *schema-*

```

Flights Dataset:
Int,String,String,String,String,String
20199,PG0320,08-09 06:55,08-09 09:15,MRV,PEE
Airports Dataset:
String,String,String,Float,Float,String
TOF,Bogashevo Airport,Tomsk,85.21,56.38,Krasnoyarsk
Duration:
24h

```

Figure 3.2: A sample configuration file required by **BigFuzz**. The file contains schemas and seed inputs for the `flights` and `airports` dataset, and a user-specified time cut-off for the fuzzing campaign.

```

Flights dataset:
-17252,P*34,4GJn50:0k0G,Zu:CS0.9-,D)N,D)N,A]i%e(
-17252,PG04,09-o0'gc:k7,zq-j01:55",D)N,D)N,]ZTed
Airports dataset:
D)N,SYhkutkl:irp7rS,Gap/ns4,1.64E9,1.30E9,JUSg3sk+

```

Figure 3.3: A test case generated by **DepFuzz** that causes a NaN exception in the program in Figure 3.1

aware mutations such as changing the numerical value, changing integers to float, adding/removing columns, or changing the delimiter. These mutations help **BigFuzz** avoid some trivial parsing errors. Although **BigFuzz** achieves 98% statement coverage in 24 hours, it is still unable to trigger the fault in [9], because to pass beyond `join` at [8], the three columns (column 0 of `airports` and columns 4 and 5 of `flights`) must have the same value to satisfy *co-dependence* constraints to exercise the UDF of `map` at [9]. Since **JAZZER** and **BigFuzz** mutate all columns independently of each other, this three-way constraint is highly unlikely to be satisfied by their mutations.

Benefits of DepFuzz. Suppose that the data analyst uses **DepFuzz** to generate new test inputs. She does not need to provide an explicit schema and simply provides the current dataset to **DepFuzz**. At the end of 24 hours, **DepFuzz** generates new inputs as shown in Figure 3.3, leading to a NaN error, reaching the faulty line inside the corresponding UDF of `map` at [9].

DEPFUZZ detects the two sets of co-dependent regions (highlighted in blue and red in Figure 3.3) and mutates them such that they can still satisfy the implicit constraints imposed by the three join operators (4, 5, and 8). The rows in blue (*i.e.*, -17252) must be equal since 8 performs a self-join. The red cells (*i.e.*, D)N) are co-dependent by equality due to join 4 and 5. Close inspection of the application execution on this input shows that variable *c* is faulty at 9 in Figure 3.1 (b). The developer spots this error on the second last line and replaces `sqrt(.1-a)` to `sqrt(1-a)` in accordance with the Haversine formula.

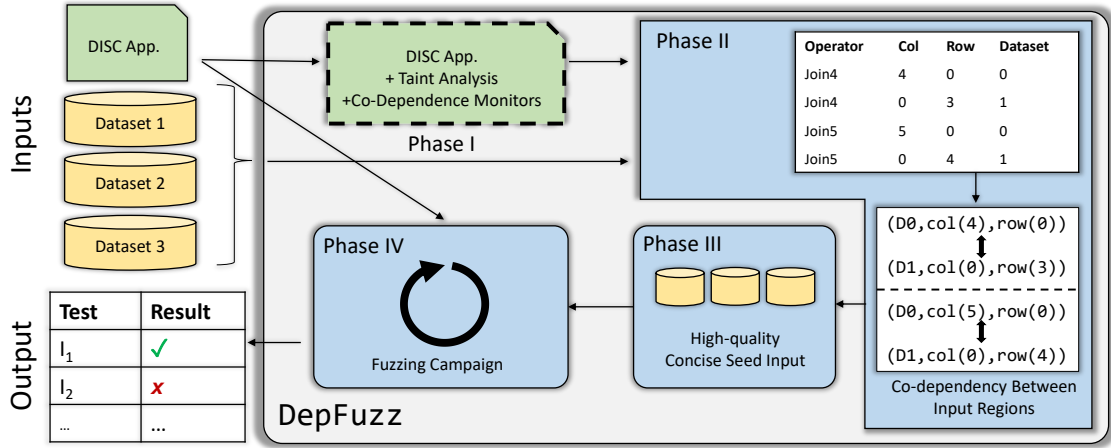


Figure 3.4: Workflow of DEPFUZZ.

3.3 Approach

The key contribution of DEPFUZZ is to detect co-dependent regions across multiple datasets and orchestrate input mutations on the co-dependent regions accordingly. In this section, we formally define co-dependence and provide details of how DEPFUZZ detects them. DEPFUZZ consists of four phases as shown in Figure 3.4. Phase I automatically instruments a given DISC application to enable fine-grained taint analysis at the level of rows, columns, and dataset IDs. This allows it to track data provenance through dataflow operators and UDFs

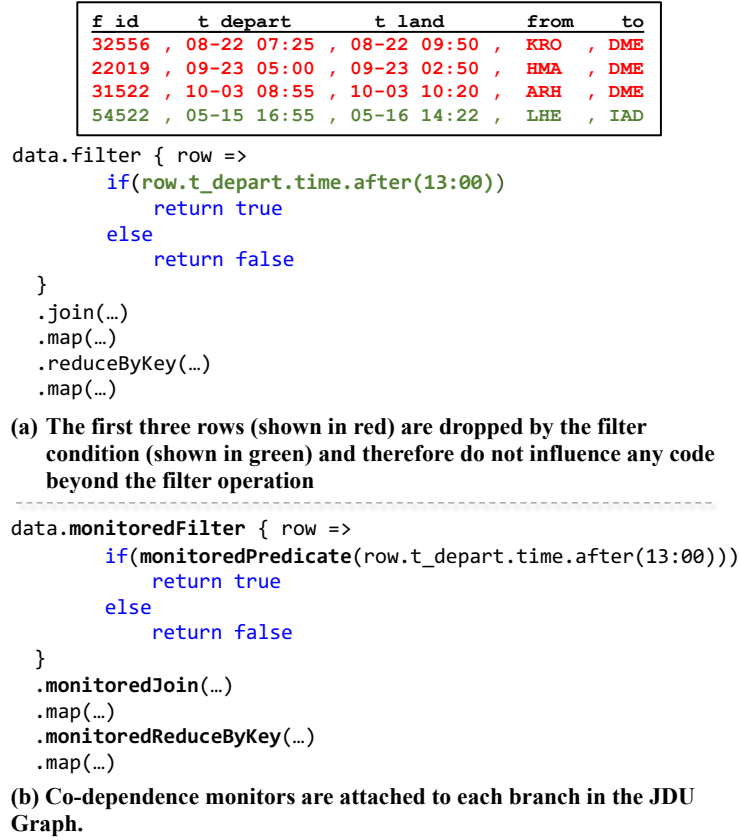


Figure 3.5: The red color rows do not participate in the join operation since the filter operation removed them earlier.

to capture co-dependence relationships. Phase II executes this instrumented program on the entire dataset to capture co-dependence constraints among multiple input datasets. Phase III leverages this provenance tracking capability to select a precise subset of rows from each dataset to use as seeds for subsequent fuzzing iterations. Phase IV then initiates a fuzzing campaign with the selected rows from Phase III and applies co-dependence aware mutations to expose deeper faults. After reaching a user-specified time limit, **DEPFUZZ** outputs a set of test inputs.

Formalizing Input Co-dependence. Co-dependence is a dependency created between multiple input regions by an operation (*e.g.*, a dataflow operator or a binary operation

that affects control flow in UDFs) that operates on such input regions. An input region is a contiguous sequence of bytes in an input dataset. We formalize co-dependence as follows. Given a DISC application, we define its dataflow graph (DFG) with two types of vertices: *operators* and *datanodes*, similar to the traditional DFG representation [63]. In the case of DISC applications, *Operators* are functions in a program that operate on data (*e.g.*, `join()` dataflow operator or `==` in UDFs). A comprehensive list of trackable operators is shown in Table 3.1. *Datanodes* represent data that propagate from one operator to another (*i.e.*, input and output of an operator). Thus, we define a DISC application's DFG, G , as

$$G = \langle O \cup N, E \rangle$$

where $O = \{o_1, o_2, \dots, o_n\}$ is a set of operators, $N = \{n_1, n_2, \dots, n_m\}$ is a set of datanodes. $E \subseteq (O \times N) \cup (N \times O)$ is a set of directed edges connecting operators with datanodes. An atomic unit of this DFG has three nodes and two edges *i.e.*, an operator with an incoming edge from an input datanode and an outgoing edge to an output datanode. Furthermore, a datanode n holds data in the form of a *byte sequence* $b_1b_2\dots b_k$. Let $D(n)$ be the set of all possible subsequences of the byte sequence in a datanode n , *i.e.*, $\{b_1, b_2, \dots, b_1b_2, \dots, b_1\dots b_k\}$. Input datasets of DISC applications are defined as S , a set of initial datanodes which are external inputs to the DFG. We combine regions in input datasets in S' , a union of $D(n)$ across all input datanodes.

$$S' = \bigcup_{\forall n \in S} D(n)$$

Finally, we characterize co-dependency among input regions as a set of tuples, C

$$C = \{(o, R) \mid R \subseteq S', \forall o \in O\}$$

The first element, o , is an operator in the dataflow graph; and the second element, R , is a

```

1 case class TaintedString(value: String, t: Taints){
2   // A Tainted String class
3   def concat(other: String): TaintedString =
4     return new TaintedString(value.concat(other), t)
5
6   def concat(other: TaintedString): TaintedString =
7     return new TaintedString(
8       value.concat(x.value),
9       union(t,x.t)
10    )
11 ... // more overloaded operators

```

Figure 3.6: Taint analysis enabled String type in Scala

subset of the regions in the input datasets that are co-dependent due to operator o . Let $I(o)$ be the incoming data to an operator o . Since co-dependence can only occur between regions of the original datasets, we must extract R from $I(o)$, which can be any arbitrary byte sequence in the incoming datanode of operator o . To extract such information, we define *monitors* that are concretely explained in Section 3.3.1.

$$M_o: I(o) \rightarrow \mathcal{P}(S')$$

where $\mathcal{P}(S')$ is the powerset of S' . A monitor, M_o , is an operator-specific function that takes $I(o)$ as input and outputs a set of byte sequences (*i.e.*, input regions) from the original input datasets considered co-dependent *w.r.t.* given operator. The precise logic behind this mapping depends on the semantics of the operators, which we capture using dynamic tainting in **DEPFUZZ**. Take for example a `==` operation in `a.substr(0,5) == b.substr(0,5)`. The monitor, $M_{==}$, should yield `{a.substr(0,5), b.substr(0,5)}`, resulting in a co-dependence tuple `(==, {a.substr(0,5), b.substr(0,5) })`. Table 3.1 lists concrete examples of operators, their monitors M , and the respective mutation strategies.

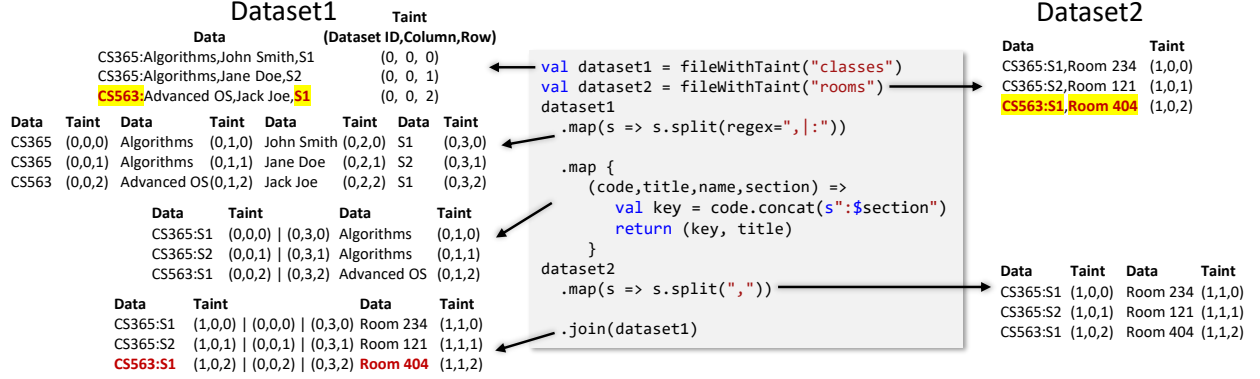


Figure 3.7: Taint propagation through a simple dataflow program. Yellow colored highlighted text is the provenance of red colored text at the bottom left table.

3.3.1 Phase I: Enabling Fine-Grained Taint Analysis

Phase I instruments a given input program to enable taint analysis and to capture co-dependence information.

Enabling taint analysis via instrumentation DEPFUZZ uses taint analysis to identify precise columns and rows in the input datasets contributing towards a specific intermediate output or a final output. DEPFUZZ first replaces primitive data types with equivalent *tainted* types. The tainted data type is a tuple of an original type and a list of offsets representing taints, $(\text{Value}[T], \text{List}[\text{Offset}])$. DEPFUZZ overrides all APIs of the original data type with taint-enabled equivalent versions to propagate their taints. For example, `concat` in `tstr1.concat(tstr2)` concatenates two tainted strings, `tstr1` and `tstr2`, and attaches a new taint with the union on the two corresponding taints. Figure 3.6 shows the implementation of `concat` in `TaintedString`. DEPFUZZ provides an instrumented version of data loading APIs that read the input datasets in tainted types instead of primitive types. Similarly, we instrument APIs for `Int`, `Float`, `Double` and `Boolean`. Apache Spark’s `textFile` API reads each row as a `String`, while the taint-analysis equivalent version reads each row as a

`TaintedString` with a row offset and a dataset ID.

Taint propagation at the level of rows, columns, and datasets Randomly mutating the entire row will likely mutate non-participating regions in the input. In Figure 3.1, the second, third, fourth, and seventh columns in the first dataset are never used by the application code. `DEPFUZZ` implements an extended taint analysis at the level of a dataset ID, a column offset, and a row offset *i.e.*, `(Value[T], List[(DatasetID, ColOffset, RowOffset)])`. For example, in Figure 3.7, `CS363:Advanced OS, Jack Joe, S1` has a taint `[0,0,2]` meaning the data is from the first dataset, the first column, and the third row. To reduce the storage overhead of attaching a tainted object, `DEPFUZZ` encodes the three offsets into a single 32-bit integer.

Co-dependence monitors In order to associate taints at the level of branches and dataflow operators, `DEPFUZZ` injects co-dependence monitors at each dataflow operator and at each branch predicate within UDFs, as shown in Figure 3.5. For example, this process replaces a dataflow operator `join` with `monitoredJoin` and replaces `if(p)` with `if(monitoredPredicate(p))` within UDFs. This co-dependence monitor injection enables `DEPFUZZ` to identify which rows and columns from which datasets directly influence individual branching decisions. Branches in a DISC application include both an explicit control predicate from an `if` statement or a `for` loop in user-defined functions and implicit branches from dataflow operators (*e.g.*, `join` and `filter`).

3.3.2 Phase II: Fine-Grained Taint Tracking

`DEPFUZZ` runs the instrumented, taint-analysis enabled version from Phase I on the original datasets. Figure 3.7 shows how data is tracked through the execution of a taint-enabled

Table 3.1: Summary of how each class of operators produces co-dependent regions in the input dataset. For simplicity, we use `row[1].col[3]` as a human-readable representation of input byte region $b_i, \dots, b_j$, where $0 < i < j < \text{size}(\text{dataset})$

Operator Class	Sample Operators	Example of Identified Constraint	Mutation strategy
Fusions	<code>data1.join(data2)</code> <code>data1.intersection(data2)</code> <code>data1.cogroup(data2)</code>	$M_{\text{join}}(\{data1, data2\})$. Possible output of M_{join} : <code>{data1.row[1].col[3], data2.row[23].col[0]}</code> <code>{data1.row[31].col[3], data2.row[52].col[0]}</code> Co-dependence tuples: <code>(==, {data1.row[1].col[3], data2.row[23].col[0]})</code> <code>(==, {data1.row[31].col[3], data2.row[52].col[0]})</code>	Any mutation applied to <code>data1.row[1].col[3]</code> must also be applied to <code>data2.row[23].col[0]</code> . If no rows with matching keys exists, select a row from <code>data1</code> and copy <code>data1.col[3]</code> to <code>data2.col[0]</code> .
Aggregations	<code>data.aggregateByKey(udf)</code> <code>data.reduceByKey(udf)</code> <code>data.groupByKey()</code> <code>data.countByKey()</code>	$M_{\text{reduceByKey}}(\{data\})$. Possible output of $M_{\text{reduceByKey}}$: <code>{data.row[2].col[2], data.row[43].col[2], data.row[63].col[2]}</code> Co-dependence tuple: <code>(==, {data.row[2].col[2], data.row[43].col[2], data.row[63].col[2]})</code>	Any mutation applied to <code>data[row=2,col=2]</code> must also be applied to <code>data[row=43,col=2]</code> . Duplicate rows and apply same mutation to key columns of duplicates.
Filters	<code>data.filter(col0 > col15)</code>	$M_{\text{filter}}(\{data\})$. Possible output of M_{filter} : <code>{data.row[23].col[0], data2.row[23].col[5]}</code> <code>{data.row[31].col[0], data2.row[31].col[5]}</code> Co-dependence tuples: <code>(>, {data1.row[23].col[0], data2.row[23].col[5]})</code> <code>(>, {data1.row[31].col[3], data2.row[31].col[5]})</code>	Any mutation applied to <code>data.col[0]</code> and <code>data.col[5]</code> must ensure that there is a true and false row for the predicate.
UDF Operators	<code>if(a.contains(b))</code> <code>if(a != b)</code> <code>if(a > b)</code>	$M_{\text{contains}}(\{a, b\})$. Possible output of M_{contains} : <code>{data.row[1].col[0], data2.row[1].col[2]}</code> Co-dependence tuples: <code>(contains, {data1.row[1].col[0], data2.row[1].col[2]})</code>	Any mutation applied to <code>data.col[0]</code> and <code>data.col[2]</code> must ensure that string <code>a</code> contains <code>b</code> for some mutations. It must also ensure it occasionally creates inputs that violate this.

program.

Co-Dependence detection Dataflow operators and UDFs pose implicit and explicit co-dependence constraints. For instance, `join` enforces an implicit constraint that, for each output row, the keys of the two joining datasets must be equal. Similarly, `if(airporta == airportb)` imposes an explicit constraint that the `airporta` and `airportb` (derived from specific rows and columns of input datasets) are equal. Co-dependence also arises between the rows of the same dataset. For example, *aggregation* operators such as `reduceByKey` and `groupByKey` result in co-dependence where one or more rows must have the same key to have an output row with the same key. Our insight is that while random mutations are unlikely

to satisfy co-dependence constraints by chance, coordinated mutations to specific row and column offsets that respect co-dependency constraints are likely to reach deeper code.

Exactly how taints are transformed into co-dependence constraints depends on the monitored dataflow operator type. For example, for `join`, the key columns of the two participating datasets must be the same (equality). For an `if` condition `if(column0 > column5)`, the co-dependence is a "greater than" relationship.

Once the instrumented application's execution on the original datasets completes, **DEPFUZZ** consolidates co-dependence information, documenting each monitor's relative position in terms of dataflow operator depth and the list of taints containing offsets at the level of rows, columns, and dataset IDs. For example, in Figure 3.1, `join` 4 has a depth of two and forms a co-dependence between column 5 of `flights` and column 0 of `airports` which act as the keys for the join. Note that **DEPFUZZ** can detect transitive co-dependence when there are overlapping constraints across multiple operators. For example, Figure 3.1 has a three-way co-dependence among three input regions since `airports` column 0 overlaps with `join` 5.

3.3.3 Phase III: Row Selection for Data Size Reduction

To speed up fuzzing, **DEPFUZZ** identifies a small subset of data rows that retain the same branch coverage as the original dataset. This reduces large-scale datasets to a set of seed inputs that are small enough for iterative fuzzing. Because the original input data may be very large with millions of rows, this step significantly reduce the scope of potential locations to mutate, increasing efficiency. For each branch, **DEPFUZZ** reduces the original input datasets to a subset of rows reaching that particular branch. It then consolidates the corresponding rows for all branches. Figure 3.5 (a) shows an example of how row selection creates a smaller, effective seed. A `filter` operator removes all flights departing before 13:00 on a given day.

Therefore, the rows highlighted in red will not influence any code beyond the first `filter`. `DEPFUZZ` thus retains only the green row in the seed input for subsequent fuzzing iterations.

3.3.4 Phase IV: Co-Dependence Aware Mutation

Phase IV performs a grey-box fuzzing campaign by designing new mutations that target various co-dependence types. The output of `DEPFUZZ` is a list of errors and test inputs revealing those errors. Different from standard grey-box fuzzing, `DEPFUZZ` prioritizes *where* to apply input mutations based on fine-grained taint tracking at the level of rows, columns, and datasets. `DEPFUZZ` designs a novel input mutation strategy that maintains co-dependency. Based on the co-dependent constraints, we categorize dataflow operators into four classes: *Fusions*, *Aggregations*, *Filters*, and *UDF Operators*. Table 3.1 summarizes mutation strategies for each class of operator.

- For *fusion operators* like `join`, `DEPFUZZ` applies the same set of mutations on the key columns of the two joining datasets to ensure equality. In Figure 3.1 (c), when `DEPFUZZ` mutates `KRO` in row 0 of the `flights` dataset, it applies the same mutations to `KRO` in row 3 of `airports`, ensuring a non-empty output for `join`.
- For *aggregation operators* like `reduceByKey`, `DEPFUZZ` duplicates a row and applies the same set of mutations on the key column of those rows, ensuring at least 2 rows in each output group. Suppose if `reduceByKey` is applied on the fourth column of `flights` in Figure 3.1. `DEPFUZZ` duplicates a row >1 times and applies the same mutation on the key of the original and duplicated rows.
- For *filter operators* like `filter`, `DEPFUZZ` applies mutation on the columns used in the filtering predicate. In case of `filter(data.col[0]) > data.col[1]`, `DEPFUZZ` can

create at least one row where this predicate can be true or at least one row where this predicate is false.

- For *UDF operators* like `map` and `flatMap` that take UDFs as arguments, **DEPFUZZ** handles control predicates in user-defined functions similar to `filter`. For example, in the case of `a.contains(b)`, **DEPFUZZ** identifies the provenance of the strings `a` and `b` as `data.col[0]` and `data.col[3]` respectively. **DEPFUZZ** then enforces the true path for this condition by embedding `b` in `a` during the mutation process.

3.4 Evaluation

We evaluate **DEPFUZZ** on four criteria: code coverage, fault detection, fault depth, and testing speed, transcribed into the research questions below.

- RQ1:** What is **DEPFUZZ**'s test coverage compared to baseline fuzzers?
- RQ2:** How many errors can **DEPFUZZ** detect compared to baselines?
- RQ3:** Can **DEPFUZZ** detect errors located in *deeper* code regions?
- RQ4:** How much overhead does **DEPFUZZ**'s instrumentation incur?
- RQ5:** Does **DEPFUZZ** achieve code coverage faster than baselines?

Table 3.2: Subject programs used in **DEPFuzz**’s evaluation. All programs represent real-world DISC use cases and are adopted from prior work. The data and code characteristics of benchmark programs are also shown. D , O , d , and R denote the number of datasets, number of operators, maximum operator depth, and total rows, respectively.

ID	Program	Description	D	O	d	R	Operators Used
P1	Webpage Segmentation [27, 40]	Find overlapping UI components on a webpage	2	9	6	1M	map, groupByKey, join, filter
P2	Customer Rewards [23]	Find the top-3 customers w.r.t purchase history	2	9	8	2M	map, groupByKey, join, filter, sortBy
P3	Flight Distance [9]	Compute distance travelled by a given flight	2	9	5	500K	map, join
P4	Bus Delays [76]	Identify bus routes that are delayed frequently	2	9	8	2M	flatMap, join, reduceByKey, filter
P5	Commute Type [48]	Identify the transportation type used on a trip	2	4	4	1M	map, mapValues, aggregateByKey
P6	WordCount [48]	Find the frequency of words	1	2	2	1M	map, flatMap, reduceByKey
P7	Delivery Faults [78]	Identify vendor sets leading to faulty deliveries	1	5	5	1M	map, groupByKey, filter
P8	ExternalCall [111]	Find the frequency of words	1	3	3	1M	map, flatMap, reduceByKey, filter
P9	FindSalary [111]	Total income of individuals earning $\leq$ \$300 weekly	1	4	4	1M	map, filter, reduce
P10	StudentGrade [111]	List of classes with more than 5 failing students	1	4	4	1M	map, reduceByKey, filter
P11	MovieRating [48]	Total number of movies with rating ≥ 4	1	3	3	1M	map, reduceByKey, filter
P12	InsideCircle [111]	Check whether the point (x,y) is in a circle	1	2	2	1M	map, filter
P13	MapString [111]	String mapping	1	1	1	1M	map
P14	NumberSeries [111]	Find the numbers whose $3n+1$ series’ length is 25	1	3	3	1M	map, filter
P15	AgeAnalysis [111]	Total number of people with different age ranges	1	3	3	1M	map, filter
P16	IncomeAggregation [48]	Average income per age range in a district	1	5	5	1M	map, mapValues, filter, reduceByKey
P17	LoanType [111]	The count of loan type within a region	1	2	2	1M	map

Benchmarks Existing dataflow benchmarks like TPC-DS [7] or Big Data Benchmark [6] are purely performance benchmarks written in SQL and therefore do not contain UDFs and non-relational dataflow operators. In contrast, the subject programs introduced by prior work on fuzzing in DISC only operate on a single dataset, omitting an entire class of operators related to real-world multiple dataset analytics. Therefore, we evaluate **DEPFUZZ** on 17 unique big data applications accumulated from *nearly all* publicly available prior work on DISC testing [48, 111], DISC debugging [100], and real-world DISC use cases [9, 23, 27, 40, 76, 78]. Collectively, our benchmark programs comprise (1) a variety of dataflow operators transformation (flatMap, map), fusion (join), and aggregation (reduce, group) operators, (2) UDFs, which are integral to DISC applications, and (3) both single and multiple input datasets, which are critical for practical data analysis.

The complete list of subject programs is shown in Table 3.2. For example, P7 [78] identifies the type of transportation used to perform the daily commutes *i.e.*, bus, car, or walk. It consolidates information on trips from two datasets to find the starting and destination zip codes, the distance traveled for the trip, and the time it took to cover this distance. Another program P2 is inspired by a commercial case study of Apache Spark [23]. It analyzes customer purchase history and rewards eligible customers (more than three instances of \$100 spending in the current year) with coupons valued proportionally to spending. This is a multi-dataset program that joins the customer information table with the purchase history table. Overall, the benchmark programs’ size is comparable to real-world industry DISC applications [107], which are in the order of hundreds of LOC but closed-sourced.

Baselines We compare **DEPFUZZ** against two baselines: (1) a state-of-art schema-aware DISC application fuzzer, **BIGFUZZ** [111]; and (2) the most advanced commercial-grade coverage guided fuzzer for the JVM, **JAZZER** [57], developed in part by Google. We compare against

these baselines because they are the state-of-the-art fuzzers for DISC applications and JVM-based applications, respectively. We use `scoverage` [90] to monitor Scala statement coverage of the applications. We provide **BigFuzz** with a seed input constructed by randomly sampling a row from the dataset, along with a schema of the dataset as in the original work. For **JAZZER**, we write interfacing code that converts the random byte stream generated by **JAZZER** into formatted datasets expected by the DISC application.

Evaluation Environment We run each tool for up to 24 hours, which is a standard experimental setting for fuzzing benchmarks, and measure statement coverage (%), cumulative error detection (%), and error depth (# of operators) in the dataflow graph of the benchmark programs. We perform these experiments on a 13-node cluster computing environment with 112 cores at 3.10GHz, 52TB storage, and 832GB memory. We run our experiments on Apache Spark 3.0 and HDFS 2.7.

3.4.1 Test Coverage Against Baseline

Figure 3.8 shows how cumulative statement coverage increases throughout the 24-hour fuzzing campaign with **DEPFuzz** and the two baselines. Y-axis represents the percentage of statement coverage achieved, and the X-axis represents the time elapsed in seconds. **DEPFuzz** significantly outperforms baselines for programs ingesting multiple input datasets and containing fusion, aggregation, and filter operators) such as P1-P5. For programs that ingest only a single dataset (*i.e.*, P6-P17), **DEPFuzz** shows slightly better performance on average in terms of coverage.

Program P1’s seventh operator is `join`, where each dataset’s key is a concatenation of three columns. Since there are six co-dependent columns related by this equality and both baseline fuzzers mutate each of the six columns independently of the others, they fail to generate

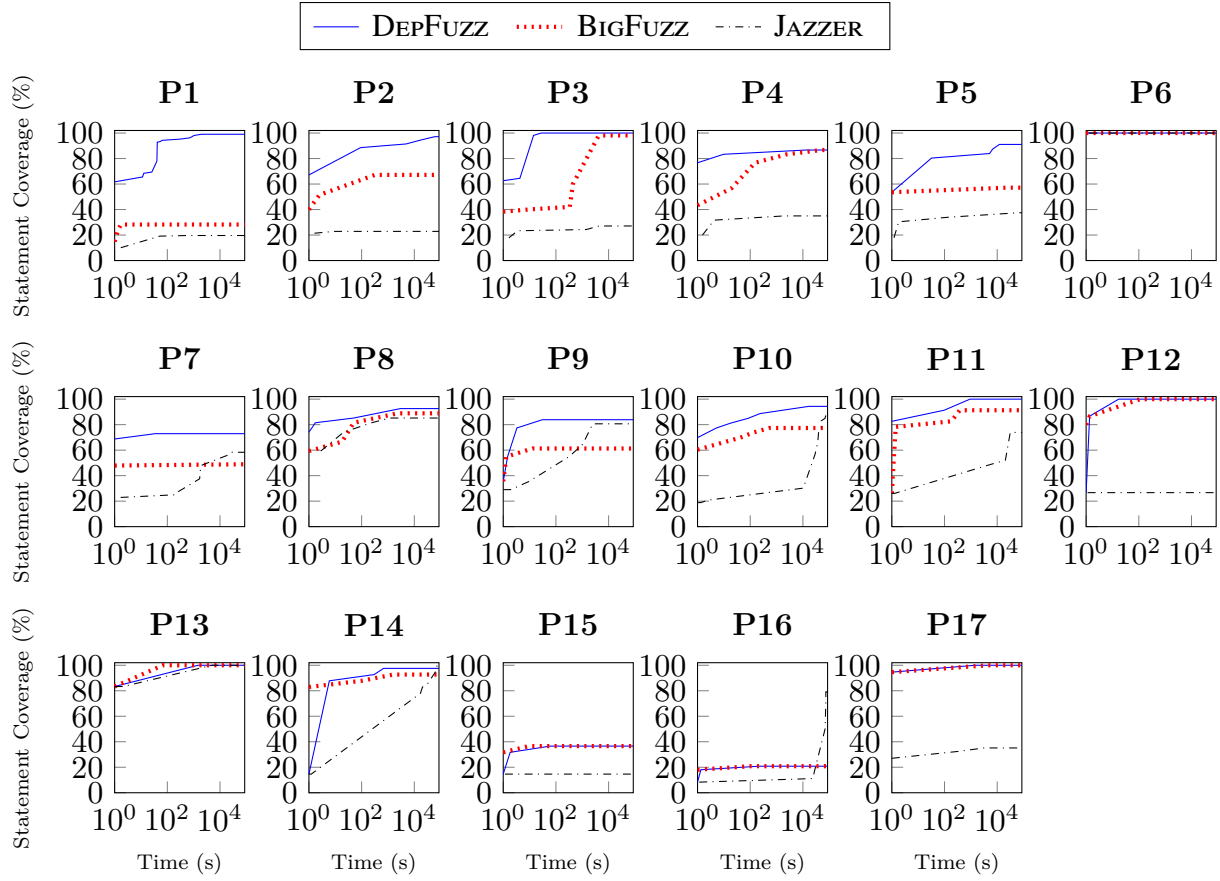


Figure 3.8: Statement coverage of three tools on 17 benchmark programs during 24 hours

even a single input with matching keys to pass this `join`. Even with its schema-aware mutations, **BIGFUZZ** only achieves 28% coverage. Similarly, **JAZZER** struggles to push beyond 20% coverage with its byte-level mutations. **DEPFUZZ** manages to capture the co-dependence between six columns created by `join`. It immediately satisfies the constraints early in the fuzzing campaign through tailored mutations for fusion operators. **DEPFUZZ** achieves 99% coverage within 24 hours of fuzzing.

In P7, we observe a drastic increase in statement coverage in the first iteration of **DEPFUZZ**, compared to the baselines. This program uses an aggregation operator, `groupByKey`, followed by `filter` that requires a minimum number of rows with the same key to exercise the code after `filter`. Mutations that randomly duplicate any row are unaware of the aggregation’s key column. Thus, baseline fuzzers do not generate the required rows to pass through `filter`. **DEPFUZZ** identifies the `groupByKey` along with the input column that influences the key. It then duplicates input rows to satisfy the joint constraint imposed by `aggregation` and `filter`. **DEPFUZZ**’s superior performance in P2-P4 can be attributed to similar reasons.

DEPFUZZ also performs better for single dataset applications P6-P17 that do not have any fusion operators (due to only one input dataset), and their average dataflow operator depth is only three. **DEPFUZZ** performs 140K fewer but more effective fuzzing iterations than baselines on average due to the higher algorithmic complexity of applying co-dependent mutations. The baseline **JAZZER** performs better in P16 because some statements in the program are only reachable on *one specific* input value. The chances of reaching such statements (*e.g.*, `stmt1` in `if(45<x<60){stmt1}`) are purely random. Thus, the technique with a higher number of iterations is more likely to reach these statements. In P16, **JAZZER** performs twice as many iterations as **DEPFUZZ**, which increases its likelihood of arbitrarily changing the input row from 90024,28,10990 to 90024,46,10990, achieving additional statement coverage. In a 24-hour fuzzing campaign, **DEPFUZZ** achieves 29% higher coverage than **JAZZER** and 13% higher

coverage than **BIGFUZZ**.

To answer RQ5, we evaluate how quickly **DEPFUZZ** achieves coverage compare to baselines by performing curve fitting with $y = mx$ as the objective function on the cumulative coverage graphs since the gradient of this line represents the average rate of gain of coverage over the course of the entire campaign. We find that **DEPFUZZ** is $1.3\times$ faster than **BIGFUZZ** and $2.1\times$ faster than **JAZZER** in terms of the coverage increase rate.

3.4.2 Fault Detection

We measure the fault detection capability of **DEPFUZZ** compared to the baselines. For this experiment, in each subject program, we inject one fault at each depth of a dataflow graph and then record the number of faults. We define a fault’s depth as the number of dataflow operators an input row has to go through before reaching a faulty statement. For example, if a fault is seeded in a UDF f , where f is an argument to n^{th} dataflow operator, the fault is seeded at depth n . For example, the fault in -Figure 3.1 has a depth of five because there are five dataflow operators before the faulty code. We count only the faults triggered from correctly-formatted inputs, as **JAZZER** generates a massive number of ill-formatted inputs that all lead to parsing errors such as `ArrayIndexOutOfBoundsException` exception from `split(",")[k]` due to missing k^{th} column in input data. Parsing errors are caused by processing ill-formatted inputs in a program. These errors normally appear in the first operation of a DISC application that takes a raw, unstructured input and parses it into individual data fields and their types *e.g.*, keys and values, similar to the first `map` applied on `dataset1` and `dataset2` in Figure 3.7. These errors do not appear if the input data format conforms to the program’s parsing logic. We evaluate fault detection on two levels: 1) the total number of unique faults detected and 2) the depth within the dataflow graph at which a fault is detected. Note that each dataflow

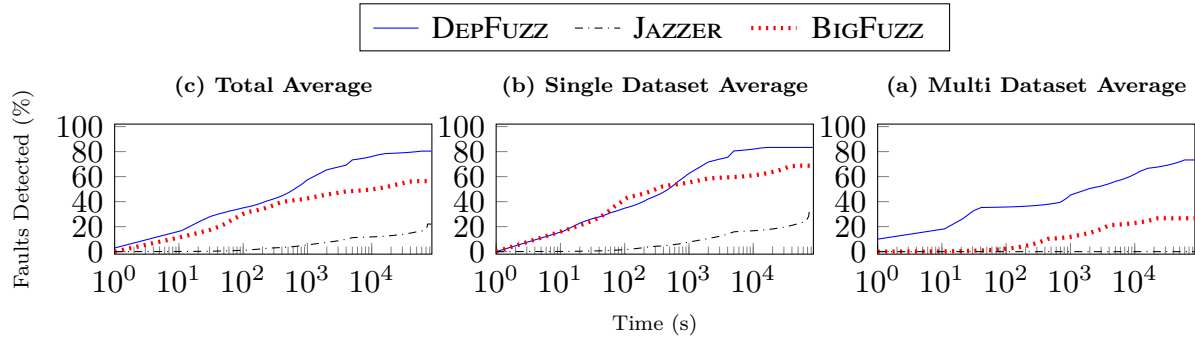


Figure 3.9: Cumulative number of faults detected during 24 hours averaged across 17 programs. (b) shows the average for P5-P17 which ingest a single dataset, and (c) shows average for P1-P4 which take multiple datasets as input.

operator takes a UDF as an argument.

Fault Injection We manually inject faults into the subject programs by randomly replacing arithmetic operators, binary operators, and constants [62]. For example, `sqrt(1-a)` becomes `sqrt(.1-a)` after injecting a fault, which can lead to NaN error. Similarly, replacing operators like `+` with `/` will inject a division-by-zero error. Prior work on Apache Spark recognizes the presence of such faults in real-world DISC applications [48]. We also add faults by employing a range check that throws `RuntimeException` if a particular column value falls within a narrow range. For example, a faulty program throws an exception if a string value in a column starts with "&%".

Fault injection is widely used in practice to evaluate new testing techniques. Automated fault injection tools such as LAVA [38] and Apocalypse [92] devise a set of principles that mimic properties of real-world faults. When injecting faults, we also follow these principles, which are as follows.

- *Rare*: The injected faults manifest for only a small fraction of possible inputs. We inject a fault that is triggered if the first column starts with the characters "&%". The number

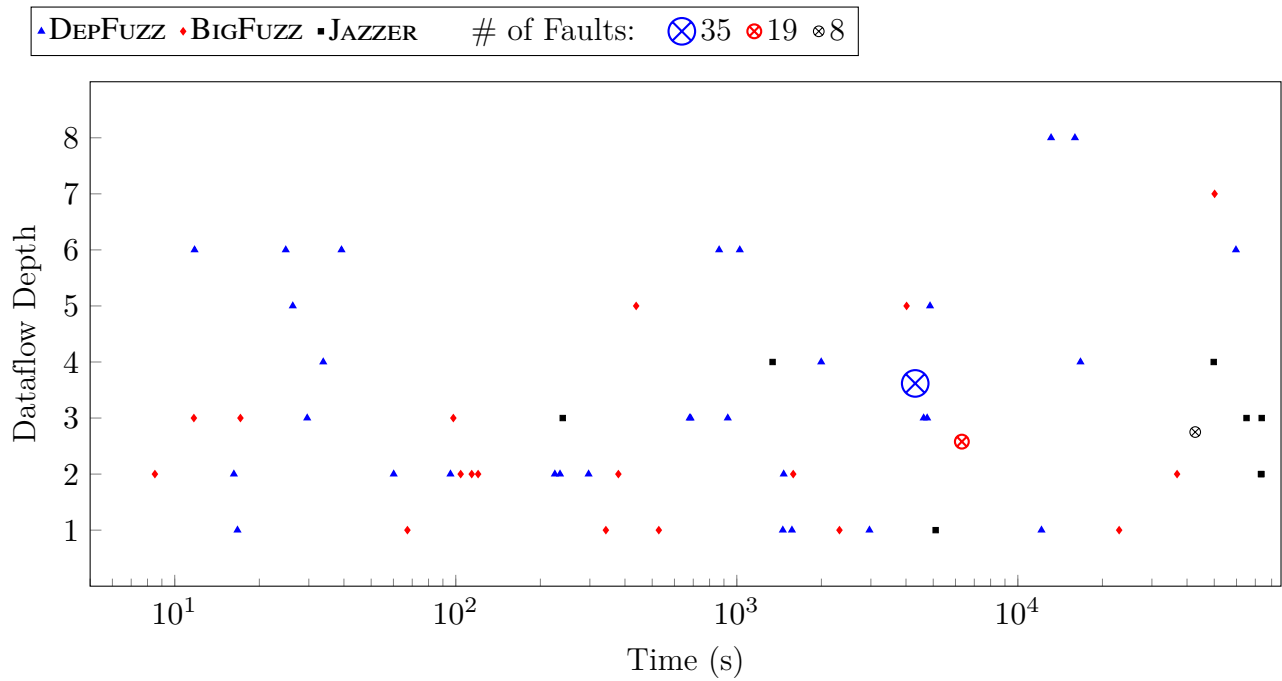


Figure 3.10: Depth vs Time for all faults detected by **DEPFAZZ** and baselines. $\otimes$ denotes centroids of each tool. **DEPFAZZ** has more points near the top left corner, which means it detects deeper faults faster than baselines. **DEPFAZZ** finds more faults than baselines.

of inputs that can trigger this fault is $\approx 256^{18}$. Assuming all inputs are equally likely, the probability of random mutations triggering this fault is ≈ 0.00002 , assuming the row length of 20 ASCII characters. Note that this is an overestimate since, depending on where the fault is injected, several other control flow and data flow criteria will need to be met for the execution to reach the injected fault, further restricting the space of inputs that can trigger this fault.

- *Uncorrelated*: Finding one injected fault neither increases nor decreases the likelihood of finding any other faults.
- *Reproducible*: The faults are deterministic and reproducible in that a single input can prove the existence of a fault.
- *Fair*: The faults are injected in locations that can be feasibly reached by an automated technique. For example, no fault is guarded by a branch that requires solving an infeasible mathematical problem, such as factoring a large integer into its constituent primes.

In total, we inject 45 faults across 17 benchmark programs. Since the location of a fault may favor certain techniques, we ensure fairness in fault injection by injecting a fault at each data processing step in every program.

Fault Detection Figure 3.9 shows the cumulative average number of faults detected on the subject programs. We report a summary of all detected program faults in Table 3.3. In Figure 3.9, the Y-axis represents the percentage of cumulative faults detected, and the X-axis represents the fuzzing duration. **DEPFUZZ** outperforms baseline techniques in terms of fault detection. For example, the majority of the inputs produced by **JAZZER** have an insufficient

number of columns, which leads to data parsing errors (*i.e.*, `ArrayIndexOutOfBoundsException` exception) after the `split` operation. Similarly, in P1, **BIGFUZZ** spends over 50% of its iterations triggering the same four parsing faults in the first UDF, causing only `NumberFormatException`. Table 3.3 lists the total faults detected by **DEPFUZZ**, **BIGFUZZ**, and **JAZZER**. On average, **DEPFUZZ** finds $3.4\times$ more faults than **JAZZER** and 84% more faults than **BIGFUZZ** due to co-dependence aware mutations. **DEPFUZZ**'s strengths in fault detection are noticeable in P1-P4 and P7, where co-dependence aware mutations help **DEPFUZZ** go past the fusion operators and reach deeper dataflow operators.

Detecting Deeper Program Faults We stratify the injected faults by their dataflow operator depth. Figure 3.10 shows a scatter plot that visualizes the depth of the faults across 17 programs. The top of the plot represents deeper, hard-to-reach faults, whereas the bottom represents faults in the initial phases of the application.

Table 3.3: Running time of the original subject program and the instrumented program with taint analysis along with total errors detected by each tool.

Program	Application Execution Time			Faults Detected		
	Original	Instrumented	Overhead	DEPFUZZ	BIGFUZZ	JAZZER
P1	9.4	36.2	3.9	7	0	0
P2	15.6	149.2	9.6	3	0	0
P3	54.8	768.4	14.0	3	2	0
P4	11.2	27.8	2.5	1	1	0
P5	17.4	174.2	10.0	1	1	0
P6	7.0	13.0	1.9	1	1	0
P7	7.0	17.6	2.5	1	0	0
P8	7.0	12.8	1.8	2	2	1
P9	5.8	6.6	1.1	2	2	1
P10	6.8	12.0	1.8	3	2	3
P11	6.6	12.0	1.8	4	1	0
P12	4.8	5.8	1.2	1	1	0
P13	5.0	5.6	1.1	1	1	1
P14	5.0	6.0	1.2	2	2	2
P15	5.0	5.8	1.2	1	1	0
P16	6.8	8.0	1.2	1	1	0
P17	4.8	6.0	1.2	1	1	0
Total Faults Detected				35	19	8

The scatter plot shows that, overall, **DEPFUZZ** finds faults that reside at a deeper dataflow depth. In P1, for instance, **DEPFUZZ** finds a total of 7 faults across three different dataflow depths (3, 4, and 6), whereas both **BIGFUZZ** and **JAZZER** are unable to find any. The plot also shows that **DEPFUZZ** is consistently faster at finding deep faults than baselines. For example,

in P3, the deepest bug is triggered by **BIGFUZZ** a little over an hour into the fuzzing campaign, whereas **DEPFUZZ** finds it within the first minute. Although the time difference is smaller for single dataset programs (P6-P17), a similar pattern can be observed. For example, in P14, **DEPFUZZ** finds the deepest bug within the first two minutes, whereas **BIGFUZZ** takes over 13 minutes. Figure 3.10 also shows the centroids with $\otimes$ for each tool. The size of $\otimes$ represents the number of detected faults. Note that the gaps between the centroids are larger than they appear due to the log-scaled x-axis. On average, the deepest faults detected by **DEPFUZZ** are 1.1 operators deeper than **BIGFUZZ** and 0.9 operators deeper compared to **JAZZER**.

3.4.3 **DEPFUZZ’s Instrumentation Overhead**

DEPFUZZ enables dynamic taint analysis in a trial execution (*i.e.*, running an instrumented program on the original input data) to identify co-dependence relationships. Note that this is a *one-time overhead for the first run* and is not a recurring overhead for each fuzzing iteration because its goal is to infer co-dependence constraints from existing data.

Table 3.3 shows the time difference between an instrumented run and an uninstrumented run on the original input datasets. For instance, in program P1, the trial execution for dynamic taint analysis takes 36.2 seconds, whereas the original program takes 9.4 seconds to process the same amount of data. This overhead is higher in programs with multiple datasets, aggregator operators, and fusion operators, P1-P5, as they introduce complex dependencies among columns and rows. These co-dependences are represented in dense taint objects (*i.e.*, RoaringBitmaps [28]). Across the 17 programs, the first instrumented run’s overhead is $1.1\times$ to $14\times$ of the first uninstrumented run. Note that this overhead is a one-time upfront cost and the rest of the fuzzing loop does not require running an instrumented version with taint monitors; therefore, in the long run, the cost of using **DEPFUZZ** becomes negligible compared

to many hours of fuzz testing. **DEPFUZZ**'s runtime overhead is on par with other taint analysis approaches on DISC applications [95].

3.4.4 Limitations of Dynamic Taint Analysis

While dynamic taint analysis helps **DEPFUZZ** extract curcial information from the program, in practice, it is incomplete. Based on our benchmark and knowledge, we construct a microbenchmark to capture patterns that cause loss of taint. Table 3.4 summarizes patterns **A–D** that result in our dynamic taint analysis being incomplete, along with illustrative examples and the corresponding microbenchmark cases.

Table 3.4: Theoretical limitations of implicit taint propagation. In each example, `t` is a tainted primitive type. `Ocurrences` denotes the number of programs in which the pattern appears.

ID	Pattern	Example	Ocurrences
A	Type casting	<code>t.asInstanceOf[String]</code>	1
B	Library calls	<code>math.min(t, 3)</code>	3
C	Pattern matching	<code>case t: Int => ...</code>	1
D	Complex types	<code>new Record(t, "init")</code>	1

These patterns represent fundamental limitations of implicit taint propagation rather than implementation oversights in **DEPFUZZ**. For **A** (type casting), explicitly casting a tainted primitive to another type using `asInstanceOf` produces a value of the target type at the JVM level, stripping away **DEPFUZZ**'s tainted wrapper in the process. Because the cast bypasses implicit conversions, the resulting value carries no provenance metadata even though it is derived from tainted input. While taint may be recoverable in specific cases, it is not possible to automatically detect and repair every cast site in general. For example, P11

casts each movie rating to `Any` using `asInstanceOf[Any]`, a common programmer hack for accommodating heterogeneous data in aggregation UDFs, and this cast causes a loss of taint. For **B** (library calls), standard library functions operate on plain JVM types rather than `DEPFUZZ`'s tainted primitives. When tainted values flow into such calls, they must be unwrapped or the call returns an untainted result, breaking the taint chain at the library boundary. Individual library functions can be replaced with tainted versions, but no finite instrumentation strategy can cover the full space of library APIs. For example, P3, P5, and P17 perform calculations that rely on the Scala `math` library. For **C** (pattern matching), type tests in a `match` expression such as `case t: Int` inspect the runtime type of the scrutinee directly rather than through `DEPFUZZ`'s tainted wrappers. Implicit conversions are not applied in this context, so a tainted primitive cannot satisfy the type test unless taint is preserved explicitly in the scrutinee; otherwise the pattern is ill-typed or forces taint to be stripped before the match. For **D** (complex types), user-defined and library complex types expose arbitrary constructors, methods, and data shapes that route values through code paths outside `DEPFUZZ`'s implicit conversion mechanism. When tainted primitives are wrapped in or passed through such types, taint metadata is not automatically propagated across the type boundary, so it is not feasible to define complete, automatic propagation rules for all possible operations. The most common way this limitation arises is through string interpolation. For example, in P7, an intermediate row is constructed using `s"$t1,$t2"` where both `t1` and `t2` are tainted values. This string interpolation desugars to a call like `StringContext(...).s(...)`, which produces a new string, but the taint information attached to `t1` and `t2` is lost in the process because the construction involves a complex type and implicit library calls. Thus, taint propagation cannot be guaranteed through such operations.

This microbenchmark can be found in the `DEPFUZZ` Github repository [1].

3.5 Chapter Summary

Traditional fuzzing is ineffective for DISC applications due to requirements to handle unstructured inputs, a lack of schema, the inability to handle multiple datasets, and their large input size. In this work, we introduce **DEPFUZZ**, a technique that uses fine-grained provenance tracking to infer complex co-dependence constraints created by dataflow operators and user-defined functions. The key insight behind **DEPFUZZ** is to orchestrate co-dependence aware mutations on multiple input datasets in concert. **DEPFUZZ** increases code coverage fast, finds more defects, and finds defects that are hard to find—29% higher statement coverage, $2.1\times$ faster, and triggering faults that are 0.9 operators deeper than the ones found by the state of the art commercial fuzzer for JVM.

Chapter 4

Natural Input Generation for Big Data Analytics

4.1 Introduction

Nearly all fuzzing techniques, including the one proposed in the last chapter, have one objective: *achieve high code coverage as fast as possible*. Every fuzzing iteration meets this objective by aggressively mutating the seed input. Due to the emphasis on incremental input mutation, fuzzer-generated inputs are often unrealistic. In DISC applications, developers are naturally reluctant to adopt such tests, as they rarely mimic real-world production data. Prior work on database (DB) testing noted that unrealistic inputs often fail to satisfy implicit integrity constraints [12]. Another drawback of fuzzer-generated inputs is that they implicitly prioritize syntactic faults rather than semantic faults located in deep, hard-to-reach regions. Albeit rarely, when fuzzing does manage to find a semantic fault, the developer may find it difficult to fix its root cause due to lack of readability.

ML-based synthetic data generators such as Synthetic Data Vault (SDV) [84] could also produce synthetic test data given a training dataset. However, these tools are not designed for the purpose of exercising different program paths, suffering from low coverage. They require an explicit schema or type information, making it unsuitable for big data analytics that ingest unstructured or semi-structured data, where each field is identified on the fly

without a predefined type. We quantify the limitation of using a synthetic data generator for testing purposes in our experiments (Section 4).

NATURALFUZZ aims to achieve high code coverage while producing natural inputs. NATURALFUZZ is built on the insight that existing datasets are themselves a rich resource for producing natural synthetic inputs. NATURALFUZZ leverages an *interleaving* mutation strategy that combines selection and splicing to generate novel inputs. It first profiles individual branches and identifies how different regions in existing input dataset influence branching decisions within a target program. It then splices out one constituent part from one input region and injects it into another, creating a brand new input that is sourced from different parts of the original input. For instance, if an application analyzes sales in November of 2022, the input dataset may contain over several years of sales data, making most of it irrelevant for testing this particular application.

To this end, NATURALFUZZ dynamically profiles individual rows. It collects the provenance of each variable used in boolean expressions. For example, for `sales.filter(year == "2022" AND month == "Nov")`, using fine-grained data provenance, it replaces `year` with `sales.col(10)` and `month` with `sales.col(11)` by tracing these variables to `sales`'s 10th column and 11th column respectively. NATURALFUZZ uses this provenance to identify whether each row in the dataset reflects a true or false branch evaluation. This branch-level profile is encoded as a bit vector. Finally, NATURALFUZZ leverages this information to efficiently identify a subset of input rows that are likely to explore new, unseen branch coverage. NATURALFUZZ only needs to run the application with the full data once for profiling, after which it performs fuzzing locally.

To evaluate NATURALFUZZ, we use a set of eight data-intensive applications from the TPC-DS benchmark [7]. We compare our technique against two available baseline techniques: Jazzer [57], a coverage-guided greybox fuzzer for Java bytecode based on LibFuzzer [94]; and

BigFuzz [111], a greybox fuzzer for DISC applications. Our evaluation aims to answer three questions about our fuzzer. Does **NATURALFUZZ** achieve more coverage than the baselines? Does **NATURALFUZZ** detect more faults? Are the inputs generated by **NATURALFUZZ** more natural? In order to evaluate the fault detection capability of the techniques, we use mutation testing. We inject faults into the benchmark programs by flipping binary operators to create mutant programs and compare outputs against the reference program.

NATURALFUZZ achieves an average of 77.6% coverage, across programs adapted from commercial benchmarks in data analytics, which is 19.9% more than **BigFuzz** and 46.5% more than **JAZZER**. It also finds $11.3\times$ more faults than **JAZZER** and $1.7\times$ more faults than **BigFuzz**. In addition to outperforming baselines, **NATURALFUZZ** adheres to the strict constraints of generating only inputs that are *natural*, which we define to be inputs that are likely to be observed in the original datasets. We use *perplexity* [87], a well-known metric in the language modeling literature to quantify the naturalness of inputs generated by all tools. We also compare our tool against a state-of-the-art machine learning-based synthetic data generator and compare the feasibility of such a tool for natural test generation compared to **NATURALFUZZ**.

The contributions of our work are summarized below:

- We are the first to incorporate the notion of naturalness and realism into fuzzer-generated data without sacrificing coverage and fault detection potential.
- Borrowing insights from the machine learning community, we use language modeling to quantify the naturalness of inputs generated by our technique and compare it to SDV, a synthetic data generator.
- We develop novel *interleaving* mutations that can generate novel, yet natural inputs. We quantify the naturalness of our inputs relative to baselines using a well-known

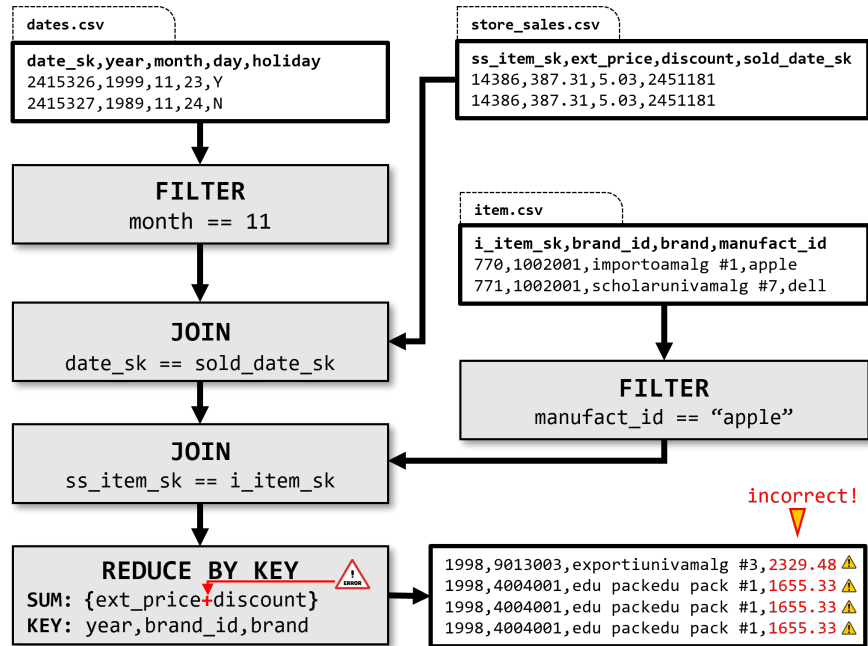


Figure 4.1: Dataflow graph of a program that computes the total sales price of items from a particular manufacturer for each year. The `+` operator in the `reduceByKey` should be a `-` and is incorrect, leading to incorrect values in the output.

metric in the natural language processing literature.

- We implement our technique in a tool called `NATURALFUZZ`. `NATURALFUZZ` is developed in Scala for Apache Spark and its key idea generalizes to other big data analytics that ingest unstructured or semi-structured data. We make our code and data available at <https://github.com/SEED-VT/NaturalFuzz.git>

4.2 Motivating Example

To motivate realistic input generation, we discuss Query 3 from the TPC-DS suite, a commercial benchmark used to measure the performance of decision support queries.

TPC-DS Query 3 finds the total revenue generated in November by all item brands of

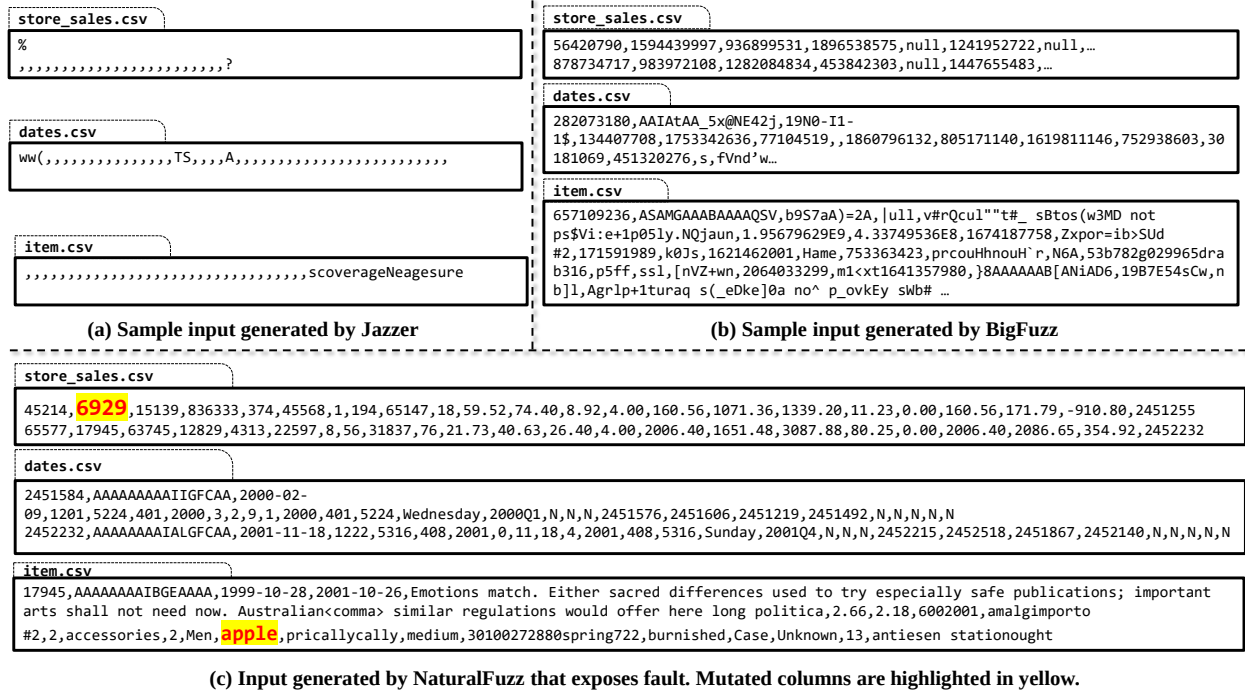


Figure 4.2: Sample inputs generated by each tool.

a particular manufacturer for a given store. Figure 4.1 illustrates the dataflow graph of this query with relevant operators. The query first reads data from three CSV (comma-separated) datasets. Dataset `dates` contains information on sales-related characteristics about each day, e.g., which quarter the date belongs to, or whether a day immediately follows a holiday or not. This table has 28 columns, although most columns are omitted for clarity. Dataset `store_sales` contains each item purchased at the store. The columns `ss_item_sk`, `ext_price`, and `discount` represent the serial key of the item sold, the extended price of the item (*i.e.*, the price after including all taxes and fees), and the amount of discount applied to the item. Dataset `item` contains available items in the store. The columns `brand_id`, `brand`, and `manufact_id` represent the brand identifier, the brand name, and the manufacturer identifier respectively.

The query first applies a filter operation on `dates` to select dates in November. Then it

performs an inner join with `store_sales` using `data_sk` and `sold_date_sk` as the joining columns from `dates` and `store_sales` respectively. This computes all the sales that happened in November of every year. Next, the program filters `item` to obtain only the items belonging to the specified manufacturer identifier such as `apple`. The resulting data is then combined with the result of the previous join via another inner join to obtain sales of products from `apple`. The resulting data is then reduced using `reduceByKey` that adds `ext_price` and `discount`. The operator uses brand and year as the key. The resulting data contains the total amount earned by the store selling each item of manufacturer `apple` every November.

Program Fault. This application is developed with an assumption that `discount` is a negative number *i.e.*, if the discount was \$4, then the column has -4 . That’s why the column `discount` is added to the sales price instead of being subtracted, as shown in Figure 4.1. When some input rows have a discount column with a positive number, the test predicate below would fail for some groups (`year`, `brand_id`, `brand`), since the discounted price is greater than the original price, as shown in the final output box and the `store_sales.csv` box in Figure 4.1.

```

1 def test_oracle(output_row, original_price):
2     if (output_row.discounted_price > original_price) false
3     else true

```

Limitations of Existing Fuzzers. We run a commercial fuzzer, `JAZZER`, on Query 3. Even after 7000 iterations, it fails to generate any input that reaches the faulty statement in the user-defined function of `reduceByKey`. `JAZZER` generated tests achieve only 58.8% statement coverage because it spends over 2000 iterations in failing to produce the required number of columns across the three datasets. At iteration 2017, it finally generates the correct number of comma-separated values. Figure 4.2-(a) shows `JAZZER` generated tests. It struggles to generate any sales data for November with the manufacturer `apple`, thus not being able to

go beyond the first filter operation. Alternatively, we run **BigFuzz**, which is a domain-specific fuzzer for Apache Spark applications. It uses *schema-aware* mutations, yet it fails to generate data (shown in Figure 4.2-(b)) that satisfy the join and filtering constraints, generating data.

Both **JAZZER** and **BigFuzz** generate unintelligible inputs. **JAZZER** mostly produces empty columns. Although **BigFuzz**'s schema-aware mutations generate correctly formatted data, they are not natural nor comprehensible to a human. Column 3 in `item.csv` should have entries in a date format. However, this column is treated as a string value, **BigFuzz** produces "b9S7aA)=2A", which has no resemblance to any date formatting. Similarly, every column in **BigFuzz** has unintelligible values making the entire input very hard for a human evaluator to analyze.

Benefits of NATURALFUZZ. **NATURALFUZZ** on Query 3 achieves 95.3% coverage in the first 13 iterations and executes the faulty statement in the `reduceByKey` operator. It outperforms the state-of-the-art and generates real-looking, meaningful data compared to meaningless inputs by baselines. The inputs generated by **NATURALFUZZ** are shown in Figure 4.2-(c). **NATURALFUZZ**'s effectiveness as a testing tool is owing to its ability to detect branches in the source code and associate each row in the dataset with a set of exercised branches. In this example, **NATURALFUZZ** locates *explicit* branch conditions in two filter operations on `dates` and `item` datasets. It also detects *implicit* branches in two join operators. Implicit branches are dataflow branches that can affect the execution of downstream code if certain dataflow conditions are not met. In the case of `join`, if neither dataset contains a matching value in the key column, no data is produced as output, and dataflow is thus terminated. **NATURALFUZZ** samples 10 rows that satisfy each implicit and explicit branch individually. It performs *interleaving* mutations using the sampled rows, randomly slicing out columns from one row and inserting them into different rows. By iteration 13, it generates the inputs shown in Figure 4.1-(c), triggering the faulty code line. The outcomes generated using these

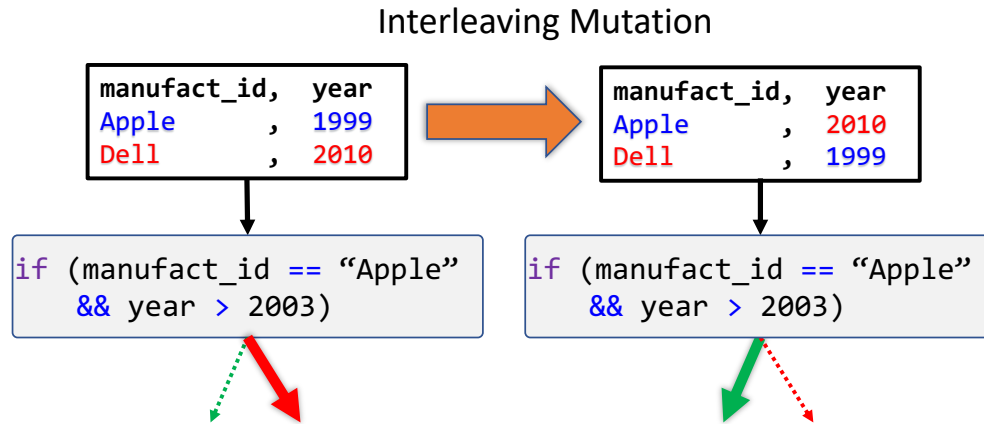


Figure 4.3: How interleaving unlocks new paths in the program

inputs indeed include rows where test oracle is violated, *i.e.*, a discounted price is greater than the original price.

4.3 Approach

There are several technical challenges in synthesizing natural inputs that also achieve high coverage and fault detection. Multiple, large-scale inputs to big data analytics exponentially increase the search space of potential targets to apply mutations. Operators such as join also create co-dependence constraints on the type and location of input mutations. Since most mutations are low-level bit or byte-level mutations, nearly all existing mutation strategies exclude natural inputs.

NATURALFUZZ is a white-box fuzzer that uses novel *interleaving* input mutations instead of traditional bit/byte level mutations to strategically create new test inputs that look natural and achieve high code coverage. NATURALFUZZ first performs a one-time dynamic path profiling to capture the application’s implicit and explicit branch predicates using taint

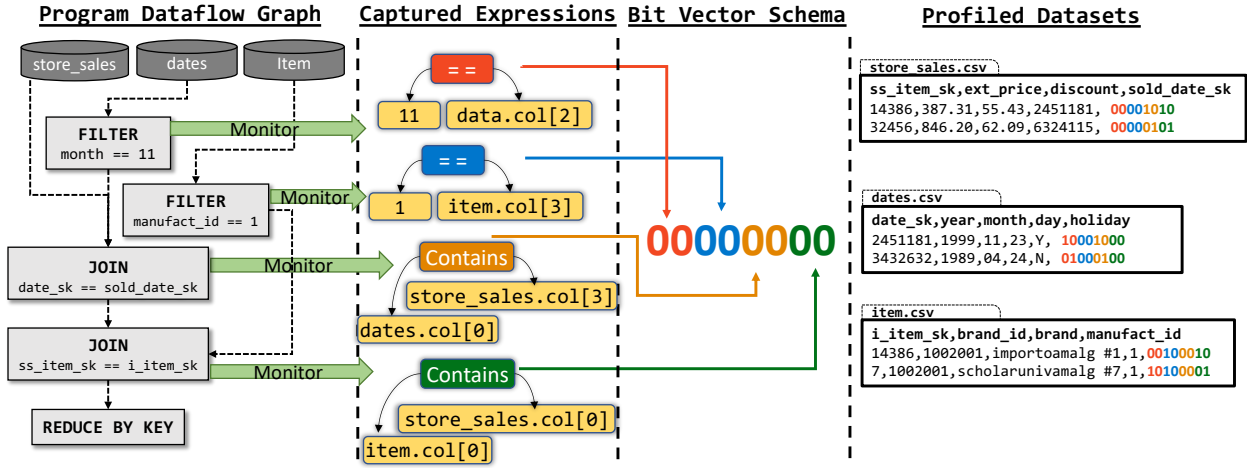


Figure 4.4: Branch predicate expressions, bit vectors, and augmented datasets constructed by NATURALFUZZ.

analysis. It then uses these predicates to map every row in input datasets to a *path vector* for each row. This information is then used to minimize the data, keeping only a small sample of rows against each program branch. In the final steps, NATURALFUZZ uses its *interleaving* mutation to generate inputs that look natural and directly influence some branching decisions in the application.

4.3.1 Interleaving Input Mutation

NATURALFUZZ's interleaving mutation mutates input datasets by substituting a piece of input row (*recipient*) with a corresponding piece from a different location in the input (*donor*). Such substitution of constituent input parts is highly effective in creating a natural input row that achieves better code coverage than the two input rows alone. This is because the recipient and donor input rows may not individually trigger new program paths. Their constituent parts may not be effective in changing branch predicate evaluations needed to exercise new paths. However, substituting constituent parts in the two rows may create a combination of

constituent parts needed to flip branch predicates and trigger a new path. Figure 4.3 shows a simple example where a branch not covered by the original data can be explored if the data is interleaved.

Finding useful interleavings of existing data is technically challenging. First, most interleavings of the data are redundant and do not improve code coverage. Second, the complex interaction of input datasets with the programs creates implicit dependencies between inputs that add another layer of constraints toward finding coverage-enhancing input interleaving. These dependencies are often formed due to dataflow operators like `join` or `reduce`. For instance, in `join`, an entry in one column of a dataset must contain in a column of another dataset, leading to an implicit branch: `if(data1.col[1].contains(data2.col[1]))`. Lastly, an exhaustive search over all interleaving is not computationally feasible. To apply interleaving mutations effectively, we must locate the precise constituent parts in each input and then measure the impact of each row’s constituent part on every branch predicate, implicit or explicit, in the program. This is the key information needed for the substitution of constituent parts across input rows that will explore new program paths.

4.3.2 Fine-grained Branch Profiling

To enable effective interleaving mutations, `NATURALFUZZ` identifies which constituent parts (rows and columns) in input datasets influence the branching decision of a given implicit or explicit branch in the program. This is the minimal information needed to model how certain input rows can satisfy a path’s constraints and exercise the corresponding path. To capture this information, `NATURALFUZZ` statically analyzes the program and locates the boolean expressions inside branch predicates in the program.

Our key idea is that since branches are primary hurdles in achieving high coverage, we can use

```

1 case class TaintedInt(value: Int, t: Taints, expr: Expression){
2 // overloaded + operator
3 def +(rhs: Int): TaintedInt =
4   return new TaintedInt(value + lhs,
5                         t,
6                         expr.add("+", rhs))
7 def +(rhs: TaintedInt): TaintedInt =
8   return new TaintedInt(
9     value + rhs.value,
10    union(t, rhs.t)
11    rhs.add("+", rhs.expr)
12  ) // more overloaded operators

```

Figure 4.5: Taint analysis enabled Int type in Scala

fine-grained knowledge of branch predicates to select only as many rows as necessary to satisfy constraints imposed by these branches. In addition to capturing the predicate expressions in every branch, we associate each variable in the expression with its source in the dataset, finding the constituent parts in input rows that directly influence the branch predicate. For example, in the expression `a > b`, `a` may be sourced from column 22 of dataset-1 and `b` may be column 3 of dataset-2. NATURALFUZZ is able to identify this and translates this expression to `ds1.col[22] > ds2.col[3]`.

We use fine-grained taint analysis to associate variables in the branch predicate with constituent parts in inputs. We extend the taint analysis engine of FlowDebug [100] to support column-level tainting and expression propagation. FlowDebug overloads data types and their constituent operations to propagate taints. We augment its taint object with a third field, resulting in a 3-tuple of the form `(Value, Taint, Expression)`. The `Expression` is a binary tree structure that tracks how the variable was constructed. For example, the `Expression` of a variable `x = a + b` would be stored as `Tree(Node(a), Node("+"), Node(b))`. Figure 4.5 shows an example of how the primitive `Integer` type is modified to support taint propagation and construct branch predicate expression. With each variable, we maintain a list of offsets from where the variable value originates and the branch predicate expression that the variable

influences in the form of an Abstract Syntax Tree (AST). This expression tree is a binary tree of operators and values that are populated as operations are applied to the variable. Each variable’s tree provides information on how to compute its value from the original dataset, owing to taint analysis. Figure 4.4 shows how the tree is developed as operations are performed. Similar to FlowDebug, NATURALFUZZ applies an automatic rewrite to use taint-enabled data types in Apache Spark.

After enabling taint analysis, NATURALFUZZ captures branch predicates by statically traversing the application’s AST. It injects a `monitor` function around branch predicates as shown in Figure 4.4. Monitor functions take the branch predicates as inputs and return their evaluation at runtime. In addition to that, since Apache Spark applications run in a distributed environment on physically separate machines, `monitors` make use of Apache Spark Accumulators [10], which are Spark’s mechanism to send small amounts of information from a worker node to the master node. We use Accumulators to send the information captured by monitors to NATURALFUZZ. After finishing the initial application run, NATURALFUZZ successfully retrieves the predicates from all implicit and explicit branches in the application and the constituent parts in the original input that for each variable involved in the predicates. Figure 4.4 shows the output of this first step.

4.3.3 Computing Path Vectors

After taint analysis, NATURALFUZZ executes the application on the original input datasets to measure each input row’s satisfiability against every branch predicate and encode this information in a bit vector, called *path vector*. A path vector is a binary number string that encodes the result of each branch evaluation for a particular row in the dataset. Figure 4.4 shows how these vectors are computed. The fine-grained knowledge of how constituent input

parts affect branch predicate evaluation can help synthesize new inputs by mixing constituent parts from different rows, resulting in an unseen path vector.

In Figure 4.4, for each branch predicate, we obtain a corresponding Boolean expression. There are a total of four branches in the program: two filters with explicit predicates; and two joins with implicit predicates. We model the implicit predicate of join as a **contains** function. This is because when an inner join is performed between two datasets, in order for the data to flow past the join, **dates** must contain at least one row where the value of the key column matches the value of the key column of at least one row in **store_sales**.

In order to compute the path vector for each row, we scan the dataset linearly and evaluate each branch expression on each row, encoding the true or false evaluation as a binary value. This creates an augmented dataset as shown in Figure 4.4. We use two bits to encode the results of a single predicate. While it is possible to encode the result of each branch as a single bit, we need two bits because certain rows may not affect the outcome of a predicate. For example, in `ds1.join(ds2).filter(ds2.col[1] < 2022)`, the expression `ds2.col[1] < 2022` does not depend on any rows from **ds1**. The role of rows from **ds1** is undefined for the predicate and thus, the expression cannot be judged to be either true or false for any row of **ds1**. Therefore, we need three possible values: true (10), false (01), and undefined (00). The value of 11 is unused. $2n$ size bit vector is needed for a program with n branches.

4.3.4 Fuzzing with Interleaving Mutation

After gathering the branch profile of each row in the input dataset, **NATURALFUZZ** initiates its fuzzing campaign that exclusively uses interleaving mutations. As with any fuzz testing approach, a good quality seed input is required for effective testing.

Algorithm 1: Seed Input Selection Algorithm

Input: A set of datasets D **Output:** A set of minimized datasets D_{min}

```

1  $D_{aug} \leftarrow \text{computePathVectors}(D)$ 
2  $D_{min} \leftarrow \{\}$ 
3  $V \leftarrow 0$ 
4 for  $d \in D_{aug}$  do
5    $d_{min} \leftarrow \{\}$ 
6   for  $\text{row} \in d$  do
7      $v \leftarrow \text{getPathVector}(\text{row})$ 
8     if  $V \neq (V \mid v)$  then
9       if  $d_j \in D_{min}$  such that  $d_j.\text{join}(\{\text{row}\}) \neq \emptyset$  then
10         $d_{min} \leftarrow d_{min} \cup \{\text{row}\}$ 
11         $V \leftarrow V \mid v$ 
12      end
13    end
14  end
15   $D_{min} \leftarrow D_{min} \cup d_{min}$ 
16 end
17 return  $D_{min}$ 

```

Seed Input selection Using the branch profile of input datasets, NATURALFUZZ filters the original dataset to obtain selected rows that assist it in achieving high coverage. The goal is to obtain a set of rows that can satisfy complex constraints in the program while being small enough for local fuzzing campaigns instead of fuzzing the entire input datasets on the cloud, which is inefficient. Selecting a small seed input is also crucial in reducing the potential locations to apply interleaving mutations. Our key idea is that since branches are primary hurdles in achieving high coverage, we can minimize such locations by reducing the input data to a small set of rows that satisfy constraints imposed by these branches.

Algorithm 1 shows how path vectors can be used for obtaining a set of rows that achieve high coverage. It selects only those input rows that increase the cumulative code coverage. The `computePathVectors` function computes the path vectors for all rows across all datasets to create the augmented set of datasets, D_{aug} . Once these path vectors have been assigned for each row in D_{aug} , we iterate over all rows of all augmented datasets, selecting rows that exercise new branch decisions, and adding those rows to the minimized dataset, d_{min} . Algorithmically, we include input when the bitwise OR of (1) the path vector of the input row and (2) the cumulative bitwise OR of the path vectors for all the rows in d_{min} (represented by

V) is never seen before. Since the path vector only tracks true or false decisions, conditions like matching keys for a join operation must be explicitly checked. In such cases, even if the path vector for the row contains true against the implicit join condition, it may not have a matching key in the d_{min} . Therefore, we check for join satisfaction with relevant datasets in D_{min} before adding it to the minimized dataset. The algorithm returns the minimized set of datasets D_{min} , which will act as the seed input.

Despite reducing data to only a necessary subset, we must ensure enough data for interleavings is available to produce novel input rows triggering new paths. This entails amplifying our donor input set by expanding on the seed input. We perform stratified sampling of the input datasets, where each stratum represents a set of rows that explore a specific branch decision in the application. For each branch, we filter the rows that are true for that branch by taking a bitwise OR with 10 at that position. We sample r rows per condition from each dataset, where r is a configurable parameter. This means that if we have b branches in the program, the donor set will contain $b * r$ rows. Note that the donor set is different than the seed input (although initiated from the seed). It is simply used as a source for generating inputs.

Interleaving Mutation Application At each fuzzing iteration, NATURALFUZZ mutates the seed input by interleaving random columns from the donor set. When applying interleaving mutation, a random column from a random input row from a donor set is selected and spliced into its corresponding column position in the seed input. NATURALFUZZ performs interleaving among input rows from the same stratum, which is also selected randomly to add variability in new input. Since the donor set and seed input is created with high sophistication, random interleaving between the donor and seed input set is enough to promote new coverage.

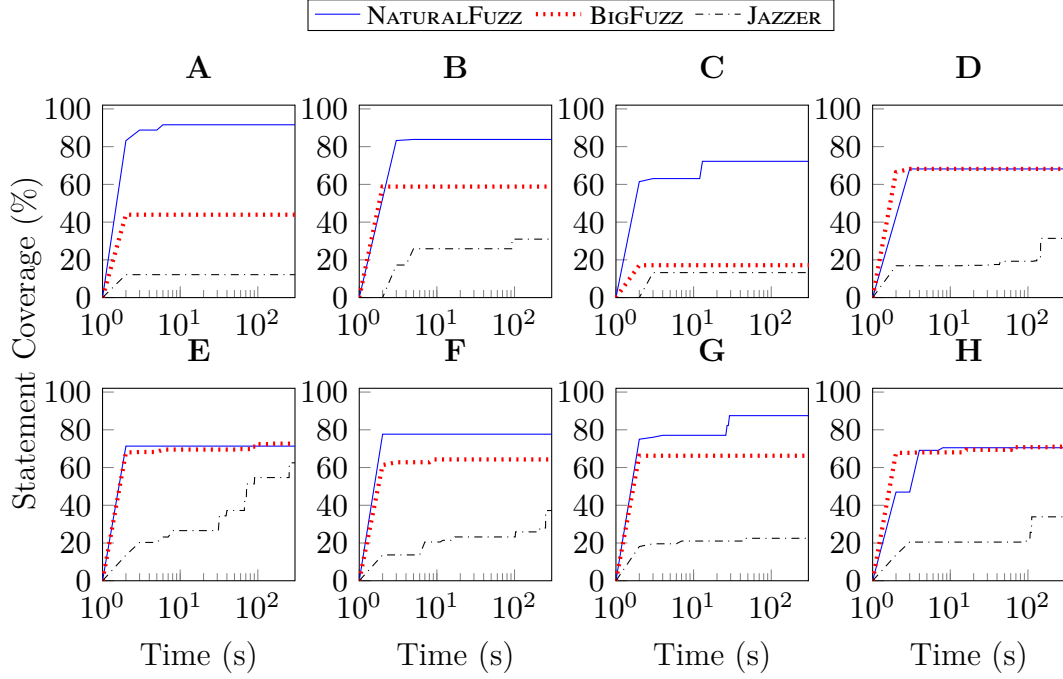


Figure 4.6: Statement coverage progress of NATURALFUZZ and baselines on eight benchmark programs.

4.4 Evaluation

Our evaluation aims to answer the following research questions:

- **RQ1:** Does NATURALFUZZ perform better than baselines in terms of coverage?
- **RQ2:** How successful is NATURALFUZZ in finding program faults compared to state of the art?
- **RQ3:** How realistic are inputs generated by NATURALFUZZ compared to baseline fuzzers?
- **RQ4:** How useful are state-of-the-art synthetic data generators for natural test input generation for DISC applications compared to NATURALFUZZ?

Table 4.1: Benchmark programs and their descriptions as taken from the TPC-DS Specification [7]

ID	Description	Datasets
A	Find customers who have returned items more than 20% more often than the average customer returns for a store in a given state for a given year	4
B	Report the total extended sales price per item brand of a specific manufacturer for all sales in a specific month of the year	3
C	List all the states with at least 10 customers who during a given month bought items with the price tag at least 20% higher than the average price of items in the same category	5
D	Compute the average quantity, list price, discount, and sales price for promotional items sold in stores where the promotion is not offered by mail or a special event. Restrict the results to a specific gender, marital and educational status	5
E	Compute the revenue ratios across item classes: For each item in a list of given categories, during a 30-day time period, sold through the web channel compute the ratio of sales of that item to the sum of all of the sales in that item's class	3
F	Report the total catalog sales for customers in selected geographical regions or who made large purchases for a given year and quarter	4
G	Select the top revenue generating products bought by out of zip code customers for a given year, month and manager	6
H	Compute the total revenue and the ratio of total revenue to revenue by item class for specified item categories and time periods.	3

Benchmarks We use a commercial benchmark suite called the TPC-DS benchmark [7], which contains SQL queries simulating real-world workloads for decision support systems. TPC benchmarks are considered a gold standard for database benchmarks and have been regularly used in prior work on database testing and debugging [79]. We present the results from eight out of the 99 TPC-DS queries. Prior work DISC application testing has mostly used simple custom applications, such as WordCount [111], for evaluations that ingest single input datasets, which is not representative of real-world analytics. Most production DISC workloads perform analytics on data that typically spans several datasets [7]. We translate the eight TPC-DS queries to Scala-based Apache Spark Applications, where every application ingests at least three datasets. Table 4.1 shows the programs used in our benchmark, including the number of datasets they use and their official description per the TPC-DS specification [7].

Baselines We compare NATURALFUZZ against two state-of-the-art fuzzers in their respective domains: (1) JAZZER, a commercial fuzzer for Java Virtual Machine (JVM) applications,

and (2) **BIGFUZZ**, the current state-of-the-art for DISC application fuzzing. We provide **BIGFUZZ** with randomly sampled rows from the datasets to act as a seed input as done in the original work. **JAZZER** requires the user to manually write *targets* for each application, which is interfacing code that converts a byte stream into a format relevant to the program. We manually provide targets for all programs. None of the two baseline fuzzers are operational on multi-datasets. To give a fighting chance, we augment the two baseline fuzzers by supporting mutations on multiple input datasets. We use a Scala compiler plugin, **scoverage**, to compute statement coverage metrics.

In addition to fuzzers, we compare **NATURALFUZZ** against SDV [84], a commercial state-of-the-art machine learning-based synthetic data generator. We seek to evaluate how useful SDV is for generating natural-looking inputs that are also useful for debugging DISC applications in practice. For a debugging use case, the size of the generated test case matters *i.e.*, the more minimal the test case the better. To this end, we create three different settings of SDV: (1) SDV-N, where 25K rows are generated by SDV and it is NOT given any schema information for the datasets; (2) SDV-S, where 25K rows are generated and SDV is given schema information; and finally (3) A variant of SDV-S, where only 50 rows are generated and SDV is given schema information instead of 25K. In contrast, **NATURALFUZZ** only generates 1.7 rows per dataset per test case.

Fault Injection We perform mutation testing [60] to evaluate fault detection capability for semantic faults and report the number of errors detected. In total, our fault injections result in 64 mutant programs. We replace every binary operator in the program with another randomly chosen binary operator. We run each tool on each mutant for a total of five minutes, which is more than the time spent on fuzzing by prior work [111]. We further eliminate any experimental variations by averaging our results over five runs. In total, we run approximately

Table 4.2: Average overhead and average total errors detected by each tool.

Program	Application Execution Time (s)			Faults Detected		
	Original	Instrumented	Overhead	NATURALFUZZ	BigFuzz	Jazzer
A	15.1	58.4	3.9	4.6	0.0	0.0
B	13.2	76.6	5.8	2.6	2.0	1.0
C	45.7	603.9	13.2	4.2	0.0	0.0
D	20.9	317.3	15.2	3.4	5.0	0.0
E	16.2	97.9	6.1	2.3	3.0	0.8
F	26.6	364.6	13.7	1.0	0.0	0.0
G	93.3	1188.6	12.8	2.3	2.0	0.0
H	14.7	79.6	5.4	0.0	0.0	0.0
Total Faults Detected				20.4	12.0	1.8

960 fuzzing jobs. We perform branch profiling steps on large-scale data on a 13-node cluster computing environment with a total of 112 cores and perform fuzzing locally on the master node, which simulates how these tools would typically be used. We run our experiments on Apache Spark 3.0 and HDFS 2.7.

4.4.1 Code Coverage

Figure 4.6 shows the progression of cumulative statement coverage over the course of the fuzzing campaign using NATURALFUZZ and two baselines. The Y-axis depicts the percentage of statement coverage attained, while the X-axis indicates the time that has elapsed in seconds. We observe that NATURALFUZZ significantly outperforms the baselines on average.

In benchmark A, NATURALFUZZ achieves 91.5% coverage compared to 43.9% and 12.1% by JAZZER, respectively. This is because A ingests four datasets that are all merged together via a `join` operator. As described in Table 4.1, the program finds people who return items more often in a particular state for a particular year. The constraints for the year and state are imposed using `filter` operations such as `dates.filter(year == 2022)`. The program also performs three inner joins to merge the four datasets, imposing a complex constraint that spans across all four datasets requiring matching keys in every column participating in the

joins. **JAZZER** struggles to meet the basic input requirement for the four datasets, resulting in very low coverage due to failures at the parsing stage. **BIGFUZZ** produces correctly formatted inputs with the correct number of columns due to its schema awareness. Still, it is unable to produce any input that satisfies the filtering constraint on the dates dataset, let alone produce any data that satisfies the constraints imposed by subsequent joins. **NATURALFUZZ** detects these constraints and produces a minimized set of datasets that contains all necessary rows to satisfy the constraints, resulting in high coverage. The best case performance for **NATURALFUZZ** out of five runs is 99.1% statement coverage, whereas the best case observed for **BIGFUZZ** and **JAZZER** is 43.9% and 12.1%, respectively. **NATURALFUZZ** outperforms the baselines on B, C, F and G for similar reasons.

There are cases where baselines perform equally well as **NATURALFUZZ**. For example, in E, **NATURALFUZZ** achieves approximately 1.0% less coverage than **BIGFUZZ**. This program checks if a particular sale falls within a given 30-day time period. Figure 4.7 shows a code snippet of this program that performs the date-time comparison. It uses Java’s `time.LocalDate` library to parse the date string from column 2 of the dataset and applies a range check for the date using the library’s `isBetween` function.

The use of the library function is problematic for the underlying taint analysis engine [100] that **NATURALFUZZ** relies on. It loses the taint whenever a library function is called. Even though the value `row.col(2)` is tainted, the variable `date` is not. The function `LocalDate` expects an argument of type `String` as its first parameter, and thus, type `TaintedString` is implicitly cast to `String`, resulting in loss of taint. Consequently, **NATURALFUZZ**’s coverage stays 1% below **BIGFUZZ**’s. The reason **BIGFUZZ** achieves 1% more coverage is because it generates an incorrectly formatted date resulting in an exception being caught at line 11. Since the **NATURALFUZZ** does not perform random mutations, it does not generate an ill-formatted date, avoiding a trivial parsing exception. The results in D and H can be attributed

```

1 import java.time.LocalDate
2 dates.filter {
3     row : Array[TaintedString] =>
4         val start = LocalDate("1999-01-01","YYYY-mm-dd")
5         val end = LocalDate("1999-02-01","YYYY-mm-dd")
6         try {
7             val date = row.col(2) : TaintedString
8             val conv = LocalDate(date, "YYYY-mm-dd")
9             conv.isBetween(start, end)
10        } catch { /* skip row */ }

```

Figure 4.7: Tainted String on line 7 is casted to String on line 8, resulting in a loss of taint. This can cause NATURALFUZZ’s expression capture to be incomplete.

to similar reasons.

4.4.2 Error Detection

We also evaluate NATURALFUZZ and baselines in terms of fault detection. Our goal is to evaluate the efficacy of the tools in detecting semantic faults rather than trivial parsing errors. To this end, we perform mutation testing and report the number of mutants killed as detected errors. We create program mutants by traversing the AST of each program and replacing binary operators with randomly selected operators. For example `a == b` is transformed to `a > b`. We replace any boolean operators in the set `{==, >, >=, <, <=, !=}` and any arithmetic operator in the set `{+, -, *, /}` with some other operator in the same set. To avoid any planting biases, we do this for every binary operator in all programs. In total, across all programs, we create 64 mutants. Table 4.2 shows the average number of faults detected by NATURALFUZZ and baselines. NATURALFUZZ significantly outperforms baselines, detecting an average of 20.4 faults compared to 12.0 and 1.8 by BIGFUZZ and JAZZER, respectively.

For example, in A, NATURALFUZZ is able to detect an average of 4.6 injected faults, whereas BIGFUZZ and JAZZER are not able to detect any fault. A computes the average number of

customer returns for a particular state for a particular year. It then multiplies this value by 1.2 and filters customers based on the criteria that `customer_returns > avg_returns*1.2`. During fault injection, this is changed to `customer_returns > avg_returns-1.2`. Since this bug appears in the code after all the data has been filtered by year and all four datasets have been joined, only **NATURALFUZZ** is able to produce data that satisfy the `filters` and `joins`, leading to the code region. In contrast, **BIGFUZZ** and **JAZZER** fail to produce data that can satisfy the `filter` and `joins`. Therefore, the filter predicate is never invoked, and the fault is never exposed.

4.4.3 Synthetic Data Generators for Testing

Although our primary baselines are fuzzers, we seek to understand the utility of state-of-the-art synthetic data generators towards the goal of generating realistic test inputs. For this purpose, we use the SDV [84], a machine learning-based synthetic data generator tool. We configure SDV to use Conditional Tabular Generative Adversarial Network (CTGAN) [108] as the tabular data synthesizer. SDV allows users to specify the data type of each column, such as numerical, datetime, or categorical. However, it has limited support for natural language text columns. Due to these strict column-type constraints, SDV is unable to train (and eventually crashes) on input rows that do not adhere to column type (e.g., `null` values). Therefore, we label each column in input datasets as categorical to prevent any potential crashes. Because CTGAN encodes all categorical columns as one-hot vectors by default, it can result in significant training overhead. To overcome this, we sample 1000 rows from the real table as the training data and train the model for the default 300 epochs.

DISC applications are often written for semi-structured and unstructured datasets, where inferring correct input schema is non-trivial, if even possible. Therefore, in order to perform

a comprehensive assessment, we evaluate SDV in three settings (1) When the input schema is available (SDV-S) and (2) when the input schema is unavailable (SDV-N). We run both SDV-S and SDV-N in a one-shot setting by training it on the original data and using it to generate 25K synthetic rows. This number is roughly equivalent to the total number of rows in all test inputs generated by NATURALFUZZ in the given time budget. We then measure the statement coverage and fault detection capability of these 25K rows. (3) Finally, in order to show how SDV performs when tasked with generating small test cases that are useful for debugging, we also use SDV to generate 50 rows per dataset instead of 25K.

Table 4.3 reports the results of these experiments. The coverage and fault columns show the average final coverage and faults detected by each tool respectively. The time column shows the total test generation time in minutes. For NATURALFUZZ, the test generation time is the sum of the time it takes to profile the program and fuzz it. For SDV, the test generation time is the sum of the total time needed to train the SDV machine learning model and the time taken to generate 25K rows for each table. All the experiments of SDV were run on a Dell PowerEdge R630 Server with 224GB RAM and 2 Intel Xeon E5-2640 v3 2.60GHz 8-core processor CPUs running Ubuntu 22.04.

Coverage In terms of coverage, NATURALFUZZ outperforms all three baselines on average, achieving an average final cumulative coverage of 77.6% compared to 74.9% and 63.8% by SDV-S and SDV-N, respectively. The high coverage achieved by SDV-S is expected since it generates 25K rows per dataset. We notice a significant drop in coverage ($\approx 17\%$) when only 50 rows are generated. Note that NATURALFUZZ achieves high cumulative coverage while generating significantly less than 50 rows per dataset (typically two rows). We observe a noticeable difference in coverage between SDV-S and SDV-N. Without the schema, SDV-N is equivalent to random sampling as it considers each unique input row as a new category.

Table 4.3: Comparison of coverage, fault detection, and generation time between SDV and NATURALFUZZ. Time is reported in minutes.

SDV-N (25K rows/dataset)				SDV-S (50 rows/dataset)		
Program	Coverage	Faults	Time	Coverage	Faults	Time
A	77.2	0.0	5.8	50.1	0.0	179.5
B	65.9	2.0	5.6	62.4	2.0	196.6
C	18.6	0.0	9.4	17.5	0.0	248.4
D	73.4	3.4	7.7	67.0	2.0	200.2
E	71.9	3.0	5.6	68.8	3.0	303.4
F	66.1	1.0	7.4	62.5	1.0	335.6
G	71.3	4.0	9.6	67.5	3.0	248.9
H	66.4	0.0	5.4	66.4	0.0	316.6
Total Faults		13.4		Total Faults		11.0
Avg Coverage		63.8		Avg Coverage		57.7
SDV-S (25K rows/dataset)				NATURALFUZZ (1.7 rows/dataset)		
Program	Coverage	Faults	Time	Coverage	Faults	Time
A	60.0	1.8	181.1	90.7	4.6	6.2
B	95.5	5.6	198.2	83.8	2.6	6.6
C	57.7	1.2	250.6	74.8	4.2	15.7
D	70.1	4.0	201.9	68.2	3.4	11.6
E	71.9	3.0	305.9	71.5	2.3	6.8
F	89.6	2.0	338.3	77.7	1.0	11.3
G	80.4	4.4	251.0	84.1	2.3	25.8
H	73.8	0.0	319.1	70.5	0.0	6.5
Total Faults		22.0		Total Faults		20.4
Avg Coverage		74.9		Avg Coverage		77.6

During data generation, its generative model uses the knowledge of categories to select one category, which happens to be an input in the training data. SDV-S does not generate completely novel input rows compared to the training dataset. However, due to its training on columnar data, where each unique entry in a column is considered a category, it produces new rows by selecting random categories from each column.

Fault Detection We observe that SDV-S, manages to detect 22.0 faults on average compared to 20.4 by NATURALFUZZ. Without the schema, it detects only 13.4 faults in total

on average. However, it is important to note that SDV is generating a single input containing 25K rows per dataset. Table 4.1 shows that benchmark programs ingest three or more datasets. Thus, SDV must generate at least 25K rows for each dataset. For example, for A, the total data generated will be 100K rows. Although it is expected for an input of this size to trigger all these errors and achieve high coverage, this is not ideal for testing. In order to test if SDV can detect these errors when generating smaller test cases, we generate only 50 rows using SDV-S, the best-performing variant. We see that the average number of total faults detected drops to 11.0, which is significantly lower than NATURALFUZZ, which generates even smaller test cases.

Limitations of SDV Our experiments show that SDV-S provides almost similar fault detection and code coverage as NATURALFUZZ. However, there are some key factors that make NATURALFUZZ a more desirable test generation tool than SDV: (1) SDV-S takes significantly longer ($22.6 \times$ more time than NATURALFUZZ) to train a model on 1000 input rows, which is prohibitive at the scale of big data; (2) NATURALFUZZ generates several minimal test cases over the course of the fuzzing campaign, each spanning only a handful of rows, typically 2 rows per dataset. In contrast, SDV must generate 25K rows at once to achieve comparable coverage and fault detection. This is infeasible for debugging since the 25K rows are too large of an input to pinpoint the error-inducing rows. (3) SDV requires a schema and configuration effort for each new program, whereas NATURALFUZZ’s performance is out-of-the-box, with no special configuration required for each program.

4.4.4 Test Input Naturalness

Although it is evident from visual inspection that the inputs generated by NATURALFUZZ are more natural than those generated by baselines, we still quantify the naturalness of the

Table 4.4: Average naturalness scores of inputs generated by NATURALFUZZ and baselines.

	Original Data	BIGFUZZ	JAZZER	SDV-S	SDV-N	NATURAL FUZZ
DistilGPT2	1.6	164.0	3327.0	2.4	2.1	2.0
DistilBERT	12.5	388.2	NaN	18.8	11.9	12.1

generated inputs w.r.t the original data. We borrow tools that are well-established in the machine-learning community to achieve this. In particular, we make use of the perplexity score [87], which is used to measure the naturalness of text generated by language models. Intuitively, a perplexity score encapsulates the probability of observing a given piece of text in a corpus. However, since perplexity is the inverse of probability, it can be thought of as how *surprised* or “perplexed” a language model is when observing a row w.r.t the distribution it has modeled. Therefore, in terms of identifying more natural inputs, *lower perplexity score reflects more natural input*. This metric is useful for us since a *natural* test input is one that is likely to be observed in the original dataset and least *surprising* for the language model that has modeled a given distribution (*i.e.*, the original dataset). Perplexity is a relative metric, so we formally define naturalness for generated inputs in the following manner: given a dataset and two generated test inputs a and b , a is more natural than b if $P(a) < P(b)$, where P is the perplexity function.

We sample 30 test input rows generated by each tool for each dataset and report the average perplexity. We employ two language models to evaluate the naturalness of the generated data: DistilGPT2, a lightweight version of OpenAI’s well-known GPT2 model [86] and DistilBERT, a lightweight version of Google’s BERT model [37]. Although architecturally similar, BERT is an encoder designed to learn the language and create general-purpose representations that can be used for any downstream task, such as text summarization. GPT2 is a decoder whose primary purpose is to generate new text. We compute the BERT score for a row by exponentiating the loss of the trained model on the row. Since BERT randomly masks tokens

from the row, the score is not stable. Therefore we average the score over 10 repetitions for each dataset.

Table 4.4 shows the average naturalness scores for each tool along with the baseline score of the original dataset. **NATURALFUZZ** achieves a low perplexity score of 2.0, which is only slightly higher than the perplexity for the original dataset, *i.e.*, 1.6. In comparison, test inputs generated by **BIGFUZZ** and **JAZZER** achieve an average perplexity of 164.0 and 3327, showing that they are unlikely to be found in the original dataset. SDV-N achieves the second-lowest perplexity owing to random sampling from the dataset. The BERT score for **JAZZER** resulted in NaNs, presumably because the loss value was very high, resulting in a NaN when exponentiated. Note that the BERT score for **NATURALFUZZ** and SDV-N is even lower than the original data, which is possible since we are working with samples, which may introduce variability.

4.4.5 Threats to Validity

We evaluate **NATURALFUZZ** on eight benchmark programs. While prior work in big data application fuzzing uses a similar number of benchmarks, it is possible that our results may not fully generalize to all possible big data analytics. We mitigate this issue by adapting programs solely from the TPC-DS benchmark, which is the most widely-used benchmark of real-world DISC workload. Furthermore, we perform a quantitative assessment of data generated by **NATURALFUZZ** and the baselines by finding the naturalness score on only a sample of 30 rows, which may not represent the naturalness of all generated data rows. We mitigate this issue by utilizing two different classes of language models to validate the correctness of our naturalness results.

4.5 Chapter Summary

In this chapter, we presented **NATURALFUZZ**, a fuzzer that achieves high code coverage as fast as possible using natural, synthetic inputs only. **NATURALFUZZ** achieves this goal through its novel interleaving mutations that mix constituent parts of different rows in the input dataset. When tested on popular database benchmark queries, **NATURALFUZZ** reaches 77.6% statement coverage compared to 46.5% and 19.9% by baseline fuzzers, **JAZZER** and **BIGFUZZ**, respectively. More importantly, **NATURALFUZZ** generates only natural test inputs when fuzzing, facilitating the downstream fault localization and debugging process. **NATURALFUZZ** is the first work that demonstrates that existing fuzzer do not need to sacrifice the naturalness and readability of their test inputs to reach high code coverage.

4.5.1 Future Outlook

Since this work was published, LLMs have matured significantly. While modern LLMs can significantly improve naturalness-oriented testing for DISC applications, there are certain tradeoffs to consider. Most importantly, despite advances in reducing compute costs, it is still cheaper and more efficient to perform simple interleaving mutations as opposed to training an LLM to produce natural data. However, **NATURALFUZZ** still assumes that some consistent structure can be imposed on the data. In cases where data is completely unstructured natural language text, **NATURALFUZZ** may not be perform any interleaving mutations due to unclear boundaries. This limitation can be mitigated by incorporating LLM-based technique to modify data in a natural way. However, producing natural data via LLMs that is consistent with production distributions yet still novel remains a research challenge.

Chapter 5

DAG-based Workload Generation for Testing Data Intensive Scalable Computing Frameworks

5.1 Introduction

Data Intensive Scalable Computing frameworks such as Hadoop MapReduce [34] and Apache Spark [14] are integral to the data processing needs of today’s distributed, stream-processing, and machine-learning systems. These frameworks expose dataflow-based APIs that let developers build processing logic as a DAG of relational and dataflow operators (e.g., `map`, `join`, `groupBy`) with UDFs embedded in these flows [51]. This dataflow-based model enables highly composable and parallelizable computations across clusters, and it underpins not only big data platforms but also machine learning systems like TensorFlow [11]. Figure 5.1 shows examples of dataflow APIs usage in various frameworks.

DISC frameworks are error-prone, and systematically testing them remains an unsolved challenge [101]. The combination of complex operator dependencies, massive data, and custom user-defined logic can lead to subtle bugs triggered only by highly coordinated workloads. To the best of our knowledge, no general-purpose DISC workload-generation

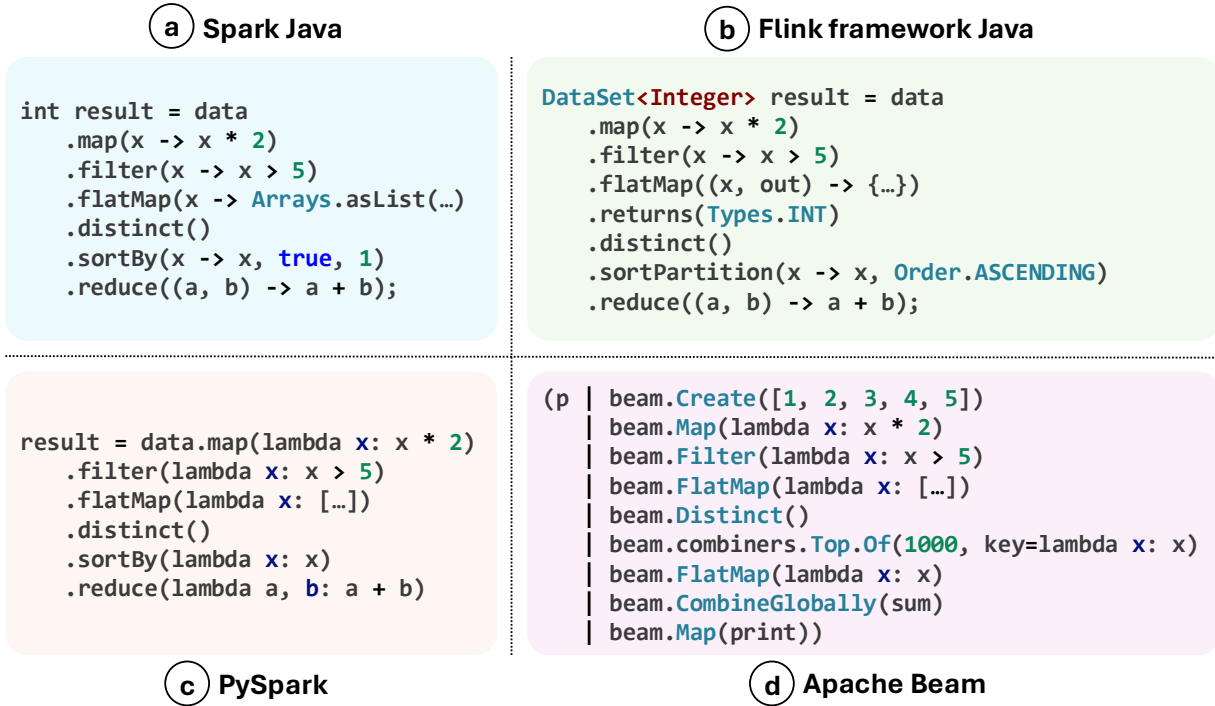


Figure 5.1: Dataflow programs in various frameworks exhibit a similar, DAG-based substructure.

framework exists today for testing DISC frameworks. While some frameworks do provide an auxiliary SQL frontend, as we will discuss in Section 5.4, recent DBMS testing work [18, 19, 42, 93, 103] are not generally applicable to DISC frameworks.

As Figure 5.1 shows dataflow APIs, by their nature, exhibits a rigid DAG-based sub-structure. This is because a distributed framework must lower the program into a workload represented as a DAG of tasks. This DAG data structure is the native input to the query analyzer, optimizer, and execution engine. Our key idea is to treat the DAG-based sub-structure of the DISC workload as a *first-class artifact* for DISC test generation and then systematically lowering it to concrete programs. By generating tests directly at the DAG level, we can decouple test generation from any particular high-level syntax or API and instead target the core dataflow abstractions shared across DISC frameworks. However, realizing DAG-based workload generation introduces several key challenges: (1) constructing a valid DAG that can

be mapped to a semantically valid program; (2) selecting appropriate operators to effectively exercise optimizer transformations; (3) populating operator parameters with meaningful values; and (4) translating the abstract DAG into executable code for the target DISC framework’s API and language.

We design our DISC test generation in four steps, addressing each challenge. First, we generate a DAG adhering to in-degree and sink constraints derived from the semantics of typical dataflow operators. Second, we associate each node, based on its characteristics, with an appropriate operator from a provided API specification. Third, we generate arguments for each chosen operator that respects the data dependencies, e.g. for a `join` operation, we generate two valid columns that are in-scope at its particular location in the dataflow path. Finally, we lower the resulting structure into a textual program that respects the topological order of operations. The details of each step are discussed in Section 5.3.

We instantiate this approach in an automated DISC test workload generation framework, **DAGGER**. **DAGGER** implements DAG-based test generation across multiple DISC frameworks, including Apache Spark [110], PyFlink [26], Polars [96], and Dask [91]. We evaluate **DAGGER** in 24-hour testing campaigns for each framework and perform an ablation study to motivate our design. However, since **DAGGER** is the first DISC test workload-generation approach, it lacks an immediate baseline. We therefore adapt a widely used SQL-based DBMS test generation tool, SQLSmith [93], to act as a baseline via the SQL frontend.

DAGGER demonstrates notably high reachability into deeper regions of the DISC framework and high framework code coverage, which enables it to find real-world, previously undetected bugs in DISC optimizers. The experiments show that, through careful DAG-based program generation, **DAGGER** achieves $7.8\times$ higher proportion of deep-reaching valid queries than the baseline. Furthermore, by generating diverse operator combinations and their parameters (including UDFs), **DAGGER**-generated test workloads cover 15.5% more lines in the optimizer

than the baseline across all the tested frameworks, on average, with the highest coverage difference of 50.0% in Polars. **DAGGER** can effectively uncover unseen bugs. For instance, it exposed a Spark query optimizer bug in the handling of nondeterministic UDFs, leading to output mismatch and state corruption. It also discovered 9 additional unseen bugs across PyFlink, Polars, and Dask, all from a 24-hour bug finding campaign. Finally, we demonstrate that **DAGGER** is highly extensible, as it can be adapted to any new dataflow API-based framework. It takes, on average, $2.6 \times$ fewer lines of code (LOC) than it takes to port SQLsmith to the four frameworks. The author, previously unfamiliar with the Dask and Polars API, took an average of 3 hours to enable **DAGGER** on these two frameworks. In summary, this chapter makes the following contributions:

- **First DAG-Based DISC Testing:** To the best of our knowledge, **DAGGER** is the first general, extensible framework for generating test DISC workloads.
- **Prototype Implementation:** We implement **DAGGER** to support four widely used DISC frameworks, including Apache Spark and PyFlink. **DAGGER** produces valid DISC workloads at a substantially higher rate, enabling deeper, high-impact testing beyond the parser logic.
- **Plan-Based Oracle for UDF Safety:** We design a novel oracle that compares optimized and unoptimized logical plans to detect violations of plan validation properties. This allows **DAGGER** to automatically flag cases like the ones where non-deterministic UDFs have been incorrectly transformed or duplicated by the optimizer.
- **Real-world Bug Discovery:** Through our experiment, we demonstrate that **DAGGER** is capable of finding real-world previously unseen bugs in open-source frameworks. Within 24 hours, **DAGGER** exposed 9 bugs across four different frameworks.

5.2 Motivating Example

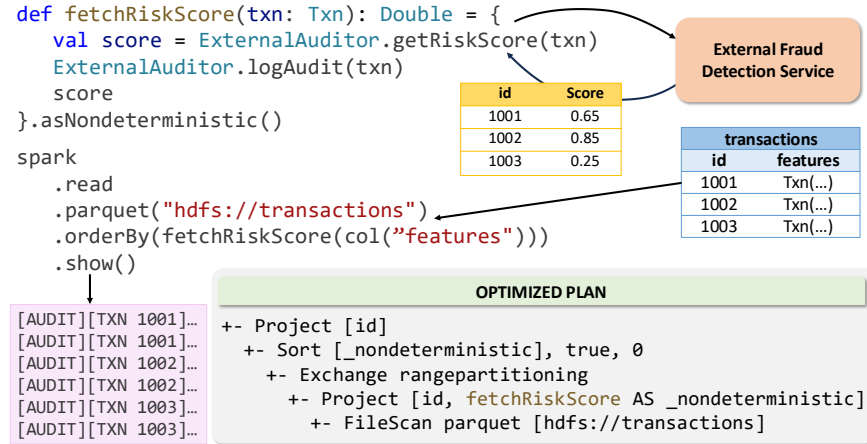


Figure 5.2: Walk through of DAGGER identified UDF non-determinism bug in Apache Spark.

We present a motivating example derived from a previously unseen bug found by DAGGER. DISC frameworks like Apache Spark allow developers to register UDFs that can interact with external systems (e.g., reading from a Redis cache), introducing both side effects and non-determinism. Spark provides a mechanism to explicitly declare such UDFs as non-deterministic using the `asNondeterministic()` method. This signals to Spark’s optimizer, Catalyst, that the UDF must not be duplicated, reordered, or otherwise transformed [16] in ways that could lead to redundant evaluations, i.e., the UDF should be invoked exactly once per input row. Non-deterministic UDFs arise naturally in production Spark workloads, e.g., systems may use UDFs to enrich input records with metadata or log fine-grained monitoring information during query execution. In such cases, redundant invocation of a non-deterministic UDF can lead to data inconsistency.

Figure 5.2 shows a simple Spark workload that demonstrates a violation of the non-determinism contract. The input dataset, `transactions.parquet` contains transaction records with ID 1001, 1002, and 1003. For each transaction ID, the table contains a `Txn` object, which is a serialized instance of a Java class. Once the dataset is loaded from HDFS,

the program uses the `orderBy` operator to sort the dataset by the *suspicion score*, which denotes the likelihood that the transaction is fraudulent. The score needs to be computed via a UDF called `fetchRiskScore`, which takes a transaction object as input, queries an external fraud detection service via `ExternalAuditor.getRiskScore()`, and logs an audit event `ExternalAuditor.logAudit(txID)`. Such a use of UDF is common in DISC [59]. This UDF is marked as non-deterministic using `.asNondeterministic()` as it makes RESTful calls to an external service, which must not be invoked redundantly. Results are printed using `show()`.

When this workload is executed on a large-scale dataset, the UDF is invoked *twice* per row: once when computing the sort key, and again when projecting the result to materialize output columns. This leads to low performance and, more importantly, incorrect outputs (i.e., wrong audit logs):

- The fraud detection service is queried twice for the same transaction, causing redundant load.
- Audit logs are written twice for each transaction ID, leading to duplicated and redundant logs as shown in Figure 5.2.
- If the external service maintains side effects (e.g., counting accesses), its counters are incorrectly incremented, breaking operational assumptions.

Such a behavior violates the Spark UDF usage contract and is a high-severity bug in Spark’s optimizer handling of non-deterministic UDFs. In a 24-hour testing campaign, DAGGER successfully detected this bug in the optimizer. Digging further into the optimizer’s behaviour, the optimized query plan appears to respect the non-determinism constraint. The plan evaluates `fetchRiskScore` once in a `Project` node and aliases the result as `_nondeterministic`, which is then used in the subsequent `Sort` operation. However, during

actual query execution, the UDF is unexpectedly invoked *twice* per row, once when computing the sort key and again when projecting the aliased column. This results in metrics being updated twice per record in the audit log. Such subtle optimizer violations are difficult to detect through conventional testing, underscoring the importance of DAG-based workloads with UDFs and plan-based oracles.

5.3 Methodology

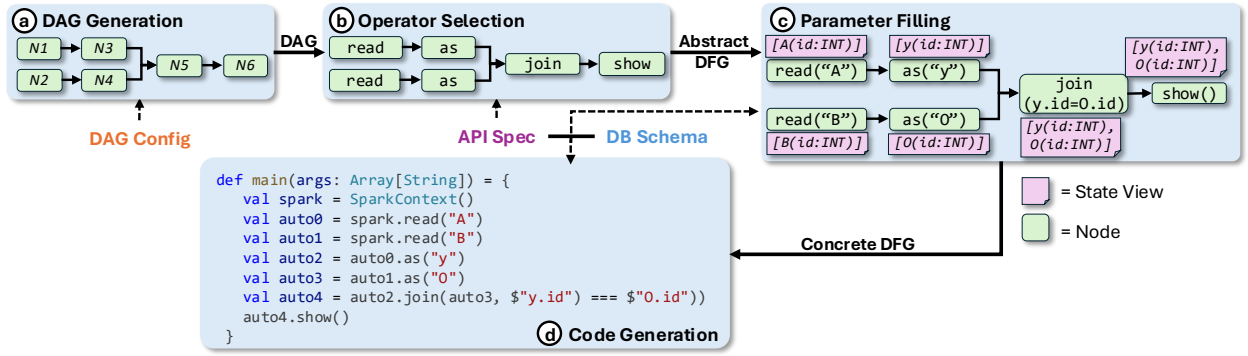


Figure 5.3: Four stage workflow of DAGGER.

This section describes the challenges in designing a DAG-based DISC workload generator and how DAGGER’s design overcomes them. Figure 5.3 illustrates DAGGER’s multi-stage test generation process to synthesize random workloads as DAGs and incrementally lower them into concrete executable DISC programs. DAGGER first generates a *context-free* DAG, where *context-free* means that each node is added to the DAG independently, regardless of the context of the surrounding nodes, representing the logical structure of a workload plan. Next, using an API specification of the framework under test, each node in the abstract DAG is assigned a specific operator (e.g., `filter`, `join`) based on in/out degree constraints, yielding an *Abstract Dataflow Graph* (ADFG) that outlines the sequence of operations without

concrete parameters. Then, **DAGGER** populates the operator parameters (such as UDFs, field names, predicates, and literals) to produce a fully *Concrete Dataflow Graph* (CDFG) while maintaining a dedicated view of the data schema (or “state”) for every node in the graph. Finally, the CDFG is translated into executable code in the target data processing framework (e.g., Apache Spark or Flink). **DAGGER** repeats these steps for every arbitrary test workload, which is then tested on the DISC framework against an oracle.

5.3.1 Stage 1: DAG Generation

First step towards generating a workload for DISC testing is to generate a random context-free DAG. Formally:

Definition 5.1 (DAG). A DAG is a directed acyclic graph $\mathcal{G} = (V, E)$ where V is a finite set of nodes which will represent operators in the next phase, and $E \subseteq V \times V$ is a set of directed edges representing data dependencies between those operators. $\mathcal{G}$ is acyclic, i.e., there is no sequence of vertices $v_0, v_1, \dots, v_k \in V$ such that $v_0 = v_k$ and $(v_{j-1}, v_j) \in E$ for all $j = 1, \dots, k$.

We base **DAGGER**’s workload generation on logical DAGs due to the nature of modern dataflow systems. In DISC frameworks, computations are internally represented as DAGs of operations; a sequence of transformations on data (e.g., the “lineage” of an RDD/DataFrame [110] in Apache Spark) naturally forms a directed acyclic graph of dependencies. By generating queries directly in DAG form, we align our test workloads with the framework’s native API, enabling more diverse plan topologies than can be expressed their the SQL front-end. Testing at the DAG level covers a broader set of operators available in modern DISC frameworks and thus exercises more of the post-parsing stages, like optimizers.

DAGGER uses a chain-based DAG construction method, which starts with a single-node DAG

Fields		Description
Operator type		The type of the operation
Return type		Return type of the operation
Parameters	Parameter Name	Name of an individual parameter
	Parameter Type	Data type of the parameter
	Required	Whether the parameter must be provided
	Default	Default value if omitted (optional)
	Values	Allowed set of values (optional)
	State-altering	Whether the parameter modifies logical state
	State-effect	Description of the state change caused

Figure 5.4: Format for the API specification.

and recursively extends it by randomly adding edges and nodes. To generate well-structured DAGs, and to align them with real-world DISC APIs, we impose two structural constraints during the generation process: 1) **In-degree constraint**: Each node may have at most two inputs. This models the fact that most dataflow operators are either unary (e.g., `filter`) or binary (e.g., `join`), and that operators with higher arity are rare since they can be expressed as compositions of binaries. These constraints can be modified in the configuration file as shown in Figure 5.3 to suit different domains. 2) **Single sink constraint**: The DAG must have exactly one terminal node. **DAGGER** adheres to these constraints and uses a random DAG generation strategy that generates from multiple source nodes to a single sink. This structure aligns well with common patterns in dataflow computation since most data analytics jobs load multiple datasets (the source nodes), apply several unary transformations such as `map` and `filter`, and merge the tables with several binary transformations such as `join` and `union`, eventually converging to a single output (the sink node). Figure 5.3-a shows a typical DAG produced by **DAGGER** containing a mixture of unary chains and binary joins, all converging at a single sink node, mirroring the structure of real DISC workloads [45].

```

{
  "withColumn": {
    "type": "unary",
    "parameters": {
      "colName": {
        "type": "str",
        "required": true,
        "state-altering": true,
        "state-effect": "column-add"
      },
      "col": {
        "type": "Column",
        "required": true
      }
    },
    "returns": "DataFrame"
  }
}

```

Listing 5.1: Example API specification for the `withColumn` operator in Spark Scala.

5.3.2 Stage 2: Abstract DFG Construction

Once a context-free DAG structure is created, **DAGGER** uses a provided dataflow API specification to assign a specific dataflow operator to each node in a *constraint-aware* manner, producing an *Abstract Dataflow Graph*.

Formally, given a context-free DAG $\mathcal{G} = (V, E)$ and a set of operators $\mathcal{O}$ defined by the dataflow API specification, the Abstract Dataflow Graph $\mathcal{G}_a$ is obtained by assigning each node in the DAG an operator from $\mathcal{O}$. This assignment is captured by a function $\lambda : V \rightarrow \mathcal{O}$, such that for all nodes $v \in V$, $\lambda(v)$ denotes the operator assigned to v . The assignment function λ is defined based on the structure of the DAG and the operator metadata provided by the API specification, as described below. Listing 5.1 shows an example specification in JSON format, and its general format is given in Figure 5.4. The goal of this stage is to map the shape of the DAG $\mathcal{G} = (V, E)$ to concrete operation types (such as scans, transformations, and joins) without yet choosing the operation’s detailed parameters. **DAGGER** bases the assignment $\lambda(v)$ on the in-degree of each node $v \in V$. Each source node is designated as a *source* operator (e.g., reading a table from HDFS or generating an initial DataFrame). Each

node with an in-degree of one is assigned a unary transformation operator such as **filter**, **project/select**, **aggregate**, or **withColumn**. Finally, each node with an in-degree of two is assigned a binary operator, e.g., **join**, **union**, or **merge**. When multiple operators are eligible, **DAGGER** selects one at random to introduce structural and semantic diversity into the generated workloads.

We drive this operator selection using the API specification, which defines the operator set $\mathcal{O}$. The specification enumerates all supported operators along with metadata: the operator name (corresponding to a method in the target API), the required number of inputs, the parameters it takes, and rules for how it modifies the dataset’s schema or state. For example, as shown in Listing 5.1, the **withColumn** operator is unary with a parameter **columnName** of type **str**. The **state-altering** and **state-effect** properties together indicate that the operator modifies the state at the column level by adding a new column (more details in the next section). An entry for **join** would have the state-effect parameter set to **false** for all its parameters since **join** does not modify state. This machine-readable specification enables **DAGGER** to remain agnostic to the dataflow engine’s internals, as one can easily extend or alter the set of operators by updating the JSON specifications. It also helps ensure that each abstract node $v \in V$ is assigned an appropriate operator $\lambda(v)$.

Figure 5.3-b shows how a context-free DAG is populated using this strategy. In the DAG generated in part (a), node $N1$ has in-degree zero and is therefore assigned the source operator **read**. Node $N3$, which depends on $N1$, is assigned a randomly selected unary operator **as** based on its single input. Node $N6$ has in-degree two, with $N4$ and $N5$ as inputs, so it is assigned a randomly selected binary **join** operator from the pool of eligible binary operators. Finally, the terminal node $N6$ flows into a sink node where the **show** operation is applied. By the end, we obtain an abstract dataflow graph where each node is labeled with a specific operation type.

5.3.3 Stage 3: State-Aware DFG Parameterization

DAGGER’s stage 3 concretizes the ADFG into a *Concrete Dataflow Graph* (CDFG) by populating all operator parameters, yielding a valid, executable DISC workload. A key challenge in this step is to ensure that the selected operators’ parameters are consistent with one another and with the evolving state of the data as it flows through the DAG. Randomly selecting column names and literal values will most likely produce an ill-defined workload, with data format mismatches and type errors. Prior work on test SQL queries generation often suffers from this issue [20]. In the case of the DISC framework, such ill-defined workloads will result in an `AnalysisException`, preventing them from reaching the optimizer and downstream components, leaving them untested. To overcome this challenge, we introduce a *state view* at each node of the DAG to guide parameter selection.

Definition 5.2 (State View). For each node v in $\mathcal{G}_a$, DAGGER maintains a finite set $\mathcal{S}_v$, representing the *state view* at v . Each element $(C, T) \in \mathcal{S}_v$ denotes an *accessible* column C together with its originating table T , where $C = (c, \tau, \mathcal{A}_c)$ and $T = (t, \mathcal{A}_t)$. Here, c is the column identifier, τ is its type, $\mathcal{A}_c$ is the set of aliases under which the column may be referenced, t is the table identifier, and $\mathcal{A}_t$ is the set of aliases associated with that table.

The *state view* of a node is the abstract representation of the data schema and other relevant properties at a particular point in the pipeline. Figure 5.3(c) shows a concrete example of how the state view $\mathcal{S}_v$ works in practice. For instance, the arguments “A” and “B” to `read("A")` and `read("B")` are generated by DAGGER by randomly choosing from a set of user-provided tables. Then, DAGGER initializes the state view for each of the `read` using the respective table’s schema. In this case, each table is a simple single-column dataset, producing a state view with a single accessible column `id` of type `INT`. The next two nodes apply `as("y")` and `as("o")`, which update the tables so that the columns may be referenced

as `y.id` and `o.id`. Since the first parameter for `as` is a `state-altering` parameter as specified in the API specification, **DAGGER** randomly generates the arguments “o” and “y” and updates the state view by updating the alias sets associated with the corresponding tables and columns. Formally, this corresponds to updating the table alias sets as $\mathcal{A}_t^A \leftarrow \mathcal{A}_t^A \cup \{y\}$ and $\mathcal{A}_t^B \leftarrow \mathcal{A}_t^B \cup \{o\}$. Once the state view has been updated, it is propagated downstream in the dataflow graph. For the node $N5$, which is a join, **DAGGER** merges the state view. The resulting state view contains the union of (C, T) pairs from both inputs. To parameterize the `join` operator, **DAGGER** generates a join condition with the help of the combined state view. **DAGGER** generates `y.id = o.id` by selecting compatible columns in the state view at $N5$. Note that **DAGGER** uses the new aliases to reference the tables. Finally, the last node contains the `show` operator, which does not take any arguments.

UDFs can also be operator parameters. **DAGGER** also allows users to auto-generate or insert their own UDFs (including UDAs, User-Defined Aggregators) in the DFG concretization step. During generation, the `state view` provides the precise UDFs signatures that contain argument types and return type for each operator that requires them. For example, consider a `filter()` function at an arbitrary point in $\mathcal{G}_a$ for which **DAGGER** must generate a UDF. A random sample of N columns $(C_1, C_2 \dots C_N)$ may be extracted from $\mathcal{S}_v$. Since the required output of the `filter` UDF is specified as a `boolean`, **DAGGER** generates the function signature $UDF : \tau_1, \dots \tau_N \rightarrow \{T, F\}$.

Generating a random UDF from its statically typed signature is a basic signature-based program synthesis problem, for which many classical and LLM-based methods exist [49, 89]. To maintain language agnosticism, **DAGGER** currently provides libraries to support generating the UDF via LLMs by leveraging a template prompt based on the type signature. However, the user is free to extend **DAGGER** to implement their own UDF generation. **DAGGER**’s library specifically leverages Claude API [13] to automatically synthesize these functions in the

target language (e.g., Scala, Python, or Java). This design allows **DAGGER** to flexibly embed diverse user-defined logic while remaining independent of any single programming language or runtime backend.

The process of filling in parameters follows a topologically sorted traversal of the DAG, ensuring that when we reach an operator node, the state of its input(s) has already been determined. When we encounter state-altering operations, we update the state view and propagate the state view forward to every downstream node via a depth-first search (DFS) from the current node to the sink. The examples below illustrate how **DAGGER** uses the state view to ensure validity:

- **Filter predicates:** When assigning a predicate to `filter`, **DAGGER** consults the input state to pick an existing column name and a compatible literal or comparison. For instance, if the input state has an integer column `A`, **DAGGER** may generate a condition `A > 10`.
- **withColumn:** For an operator that adds or modifies columns, like `withColumn` in Spark, which adds a new column based on the input expression, the state view allows us to choose a unique new column name and define its expression using existing columns. For example, if the input has columns `A` and `B`, **DAGGER** may set `withColumn("C", f(A))` where `f` is a built-in function or a UDF (e.g., `A + 5`). It then updates the current state view to include the new column `C` and propagates it to the sink.
- **Renaming (aliasing) with as:** Some frameworks allow assigning an alias to a dataset (e.g., Spark's `df.as("alias")`). **DAGGER** propagates alias information in the state so that if a later `join` refers to this dataset, it can qualify column names with the alias to avoid ambiguity. For example, if one branch of a DAG is aliased as `"T1"` and another as `"T2"` and then they are joined, **DAGGER** ensures the `join` condition follows a pattern

like $T1.X = T2.X$ on a common column X , rather than two unqualified X s.

- **Join conditions:** For a binary `join` operator, `DAGGER` verifies that the two input states share at least one common attribute type; if none exists, the DAG is discarded. Otherwise, `DAGGER` selects one overlapping column from each side as the join key (e.g., an equi-join on `key`), and the output state is the union of both input schemas (with duplicates resolved via qualifiers or by dropping one side’s duplicate per the API). The join type is chosen from the four values supported by the API spec: inner, outer, left, and right.

State-aware approach helps avoid the “short-circuit” problem noted in prior random-testing research, where a query might be trivially unsatisfiable or error-prone [20].

5.3.4 Code Generation for Target Engines

After constructing the Concrete DFG, `DAGGER`’s last step is to materialize it as an actual workload in the target DISC framework’s API (e.g., Spark). `DAGGER`’s code generation is modular and extensible, allowing different back-end languages or engines to be targeted with minimal changes. Currently, we support code generation module for Spark Scala, PyFlink [26], Polars [96], and Dask [91].

During code generation, `DAGGER` traverses the CDFG, emitting code for each node. We first produce code for all source nodes, which typically involves reading input data or creating initial RDD/DataFrames. For example, if a source node represents a parquet file stored in HDFS, the Scala codegen emits `val df0 = spark.read.parquet("hdfs://path/to/df")`. For each intermediate node, the emit function provides references to its parent nodes, the schema, and a `DAGGERCONTEXT` object that can be used to access utility functions and state information provided by `DAGGER`. We ensure that each node’s resulting DataFrame is

assigned to a unique variable, so that it can be referenced by downstream code. The single sink node’s dataframe (e.g., `dfN`) represents the final result; **DAGGER** can finish the program by either displaying it, e.g., `dfN.show()`, or writing it out, depending on the use case for testing.

5.3.5 Oracles for Bug Detection

DAGGER employs complementary oracles to detect bugs in dataflow systems. First, a differential oracle monitors for exit status mismatches between optimized and unoptimized executions e.g. crashes, runtime exceptions, and other catastrophic failures that occur during query compilation. Beyond crashes, a plan-based oracle, described in Section 5.2, can be used if the DISC application offers a non-deterministic UDF contract. This oracle performs static analysis by comparing the unoptimized and optimized logical plans to detect semantic violations without requiring query execution. This oracle is particularly effective at identifying subtle correctness bugs, such as when non-deterministic UDFs are incorrectly duplicated or transformed by the optimizer, or when trait metadata becomes desynchronized from the actual schema after projection operations. By inspecting plan structure rather than query outputs, the plan-based oracle can surface violations that would be difficult or impossible to detect through conventional output comparison, especially for non-deterministic operations or operations with side effects. Together, these oracles enable **DAGGER** to detect a broad spectrum of bugs ranging from obvious crashes to subtle semantic violations that compromise correctness.

5.4 Evaluation

We currently support testing for four most popular DISC frameworks (Spark, Flink, Dask, and Polars) in **DAGGER**, which can also be easily extended to additional frameworks. This evaluation focuses on the optimizers of these frameworks as a common testing target, as bugs in optimizers may result in incorrect rewrites, over-aggressive rule applications, and improper treatment of operators with side effects or non-determinism, but are difficult to test fully due to the expressive and dynamic nature of modern dataflow APIs. We ask the following research questions.

- **RQ1: Optimizer Reachability (Ablation study).** Does **DAGGER**'s design improve chances of a generated test to reach deeper components of the DISC framework?
- **RQ2: Optimizer Coverage.** Once **DAGGER** reaches optimizers, how extensively does **DAGGER** test its optimizer logic than baselines?
- **RQ3: Bug Detection.** How effective **DAGGER** is in uncovering new, deep semantic bugs in DISC frameworks?
- **RQ4: Generalizability** What is the effort of adapting **DAGGER** to a new dataflow framework?

Baseline Selection. There is no direct automated testing approach yet for DISC frameworks. The closest related work comes from the adjacent DBMS domain. While it is technically possible to apply SQL generation tools in an ad-hoc manner as we did, their tight coupling with SQL syntax poses three issues for their use in DISC testing. First, it limits their ability to generalize when SQL frontends are unavailable or evolve quickly. For example, Dask [91], a popular DISC framework, does not natively support SQL [32], relying

on external support which is no longer maintained [31]. Second, it limits their ability to reach deeper optimizer logic and execution engine that operates on lower-level intermediate representations, which require a complex combination of operators and user-defined functions. A general dataflow API (e.g., PySpark) often extends beyond SQL, supporting imperative manipulations, external state interactions, and complex UDF logic that pure SQL fuzzing cannot express. For instance, `mapPartitions` operator in DISC frameworks enables maintaining partition-local state, a capability essential for distributed machine learning workloads (e.g., loading a model once per partition [98]) but fundamentally incompatible with SQL’s stateless, row-oriented evaluation model. Third, since SQL is not the primary frontend of these frameworks, its support often lacks basic SQL features e.g. as of the writing of this thesis, Polars still does not support multiple tables in the `WHERE` clause. Recursive CTEs are still unsupported in all frameworks we tested, except Spark, for which Databricks only recently added support [33]. Dask and Polars do not support correlated subqueries.

For our SQL baselines, we consider SQLSmith [93], a well-known SQL synthesis tool for DBMS testing; while not natively applicable to Spark, Flink, Dask, or Polars, we incorporate it via an off-the-shelf adaptation [74] that generates SQL queries by using SQLSmith as a library. We also consider SQLancer [18], a state-of-the-art oracle-based DBMS testing tool, which operates on a fundamentally narrower scope than our approach: it targets specific correctness properties (e.g., whether a query returns correct results under a given oracle) rather than broader optimizer coverage, crash detection, or UDF determinism. Currently, it only has nascent Spark SQL support, restricted to a TLP oracle [88] for the `WHERE` clause. Unfortunately, SQLancer is currently unable to sustain a 24-hour run on Spark. It exhausts all JVM heap memory only after ≈ 0.5 hours due to Spark’s memory-intensive runs, illustrating the challenge of testing DISC frameworks.

Experiments are run on a machine with 8 cores, 4TB storage, 3.10GHz CPU, and 48GB

memory on CentOS 7. We use Spark 3.5.0, Flink 2.0.0, Dask 2026.1.1, and Polars 1.37.1.

5.4.1 Optimizer Reachability Rate (RQ1)

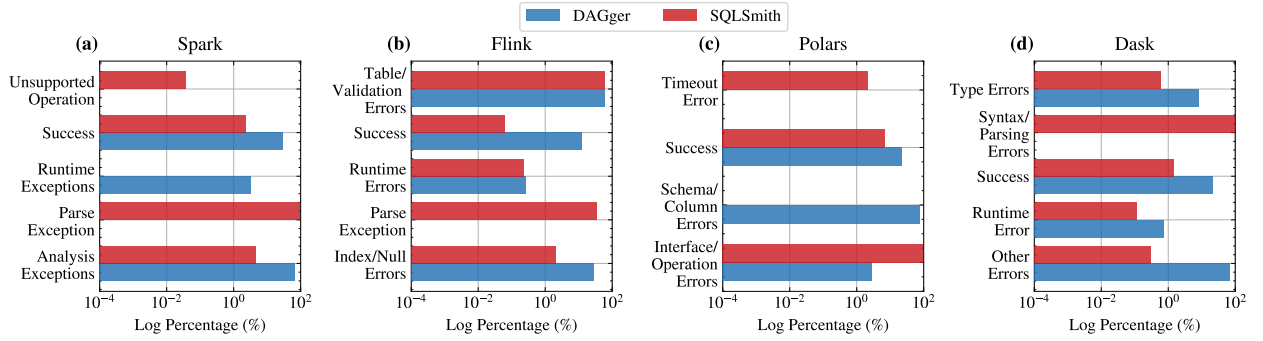


Figure 5.5: Error distribution across all frameworks.

We first ask, does **DAGGER**’s design choices help the generated DISC workloads reach deeper regions of DISC frameworks, specifically optimizers? To reach deeper regions in the DISC systems, a test workload must pass the parsing and analysis stages without errors to reach the query optimizer, code generation, and finally execution stages. For a test workload to pass the initial parsing and analysis stages of DISC, it must satisfy the target API syntax, schema constraints, type compatibility, and scoping rules. We thus consider DISC workloads that pass the initial syntactic and semantic checks as valid workloads and those that do not pass as invalid. A higher number of valid workloads naturally provides better access to deeper stages of DISC than invalid workloads, which mostly test the parsing logic.

The design of **DAGGER** is critical for achieving a high rate of valid queries. To evaluate the contribution of each design choice, we create two ablations. The first, **DAGGER-N3**, disables state-aware parameterization (Stage 3, Section 5.3.3). The second, **DAGGER-N2N3**, additionally disables constraint-based operator selection (Stage 2, Section 5.3.2). Disabling Stage 1 is infeasible, as doing so would cause an explosion in graph complexity. Here, *disabling*

a stage means replacing it with a trivial alternative: state-aware parameterization is replaced with random parameterization, and constraint-based operator selection is replaced with random selection.

DAGGER achieves an average validity rate of 22.7% across all tested frameworks, which we attribute to its state-aware parameterization and constraint-based operator selection. Disabling state-aware parameterization (**DAGGER-N3**) reduces the validity rate to 11.4%, and further disabling constraint-based operator selection (**DAGGER-N2N3**) reduces it to 6.1%, demonstrating the contribution of each design stage. Note that **DAGGER**'s search space includes substantially more complex and expressive queries derived from DAGs, containing UDFs, and unconstrained by a grammar. By comparison, SQLSmith achieves only 2.9% validity, as 94.5% of its queries fail the parsing stage and never reach the optimizer.

Having established the validity rates of **DAGGER** and its ablations, we now examine the nature of the errors that do occur. Figure 5.5 presents the distribution of errors caused by DISC applications from **DAGGER** and queries from the baseline. The invalid DISC workloads generated by **DAGGER** are mainly due to non-standard framework-specific behavior. For instance, Spark allows tables to contain duplicate column names, which can lead to ambiguity during query analysis. If a query attempts to reference a column name that appears multiple times, Spark raises an analysis exception. These issues could be mitigated by restricting column selection to source nodes that have a unique, unambiguous path to the current node in the DAG, ensuring the column's origin is clear and its reference resolvable. Additionally, certain frameworks may have structure-level behaviours that are unique e.g. in Apache Flink, applying the `limit()` operator consecutively (`df.limit(10).limit(2)`) would throw an Exception; however, this pattern is legal in Apache Spark. While it is possible to detect and avoid these patterns for a particular framework, we strike a balance between generation complexity and speed so our implementation does not verify every structural property.

Overall, **DAGGER**'s DAG-based test generation achieves better reachability by targeting the native input to optimizers, deriving tests from semantic dataflow properties rather than syntactic grammar rules. In contrast, most **SQLSmith** queries result in parse errors, an inevitable consequence of targeting SQL frontends of DISC applications, which undergo frequent breaking changes (e.g., Spark feature releases every six months [15], each with a comprehensive migration guide [5]).

5.4.2 Optimizer Coverage Analysis (RQ2)

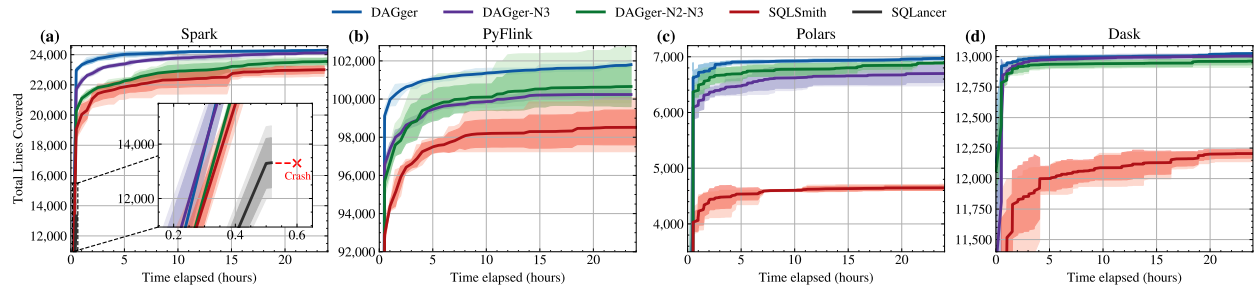


Figure 5.6: Line Coverage vs Time graphs for five trials of 24-hour fuzzing campaigns. The colored bands show one standard deviation, and lighter shades denote the min and max.

We evaluate **DAGGER**'s effectiveness by measuring the DISC framework's code covered by the tests it generates during 24-hour fuzzing campaigns. For Spark, Flink, and Polars, we measure only the optimizer's instruction coverage, as their implementations are neatly organized into separate packages and are extensively documented. For Dask, we measure full framework coverage, as its optimization logic is coupled with the framework code, with no clear package-level boundary. We also report the distribution of coverage across different optimizer subpackages for Spark and Flink, highlighting which components are stressed by each approach (**DAGGER** and baseline). For brevity, we will use "line" coverage for both line and instruction coverage from hereon. We utilize JaCoCo [39] to measure coverage for Flink and Spark. We use `llvm-cov` [71] for Polars and `coverage.py` [80] for Dask.

Figure 5.6 shows the cumulative line coverage over a 24-hour campaign, averaged over five independent trials for all frameworks, totaling ≈ 2000 hours of measurements. For instance, Figure 5.6-a compares line coverage achieved over time by **DAGGER** and **SQLSmith** for Spark. We measure coverage for Spark’s catalyst optimizer. This scope focuses on the rule-based optimizer in Spark, a critical component of the query planning pipeline. Each curve represents the mean coverage across five independent runs ($n=5$), with shaded bands indicating one standard deviation and the outer envelope marking the observed min–max range.

DAGGER exhibits consistently tighter deviation bands, reflecting markedly more stable and predictable performance across runs. This reliability arises from the specification-guided DAG generation stages in **DAGGER**’s synthesis process, which constrain the search space to semantically valid and execution-relevant DISC workloads. The ablations **DAGGER-N3** and **DAGGER-N2N3** exhibit progressively wider deviation bands and lower overall coverage, consistent with their reduced validity rates. Disabling state-aware parameterization (**DAGGER-N3**) and constraint-based operator selection (**DAGGER-N2N3**) increasingly relaxes these constraints, introducing more invalid workloads and consequently more variable coverage progression. Interestingly, for PyFlink and Polars, **DAGGER-N2N3** marginally exceeds **DAGGER-N3** in coverage despite its lower validity rate, which we attribute to the reduced per-query generation overhead from bypassing both constraint-aware stages; the resulting higher query throughput can compensate for lower precision in these frameworks. In fact, **DAGGER-N2N3** iteration count was an average of $2.4\times$ higher than **DAGGER-N3** across the four frameworks.

In contrast, **SQLSmith**’s purely random generation strategy on SQL grammar produces wide fluctuations in coverage progression, particularly during early execution, owing to its low validity rate. This difference also explains the sharper initial rise in **DAGGER**’s coverage curves relative to **SQLSmith**. A larger proportion of generated programs successfully execute and contribute to coverage from the very start, whereas **SQLSmith** expends substantial time

producing syntactically or semantically invalid queries. This is illustrated by the coverage gap starting out wider and slowly becoming narrower, e.g., the average coverage increase at the 5-hour mark is 2.2 percentage points more than the final coverage increase. Notably, the coverage gap is more pronounced for Polars and Dask than for PyFlink and Spark, largely because Polars and Dask have less mature SQL frontends, limiting SQLSmith’s effectiveness on them compared to the more established, feature-rich Spark and PyFlink.

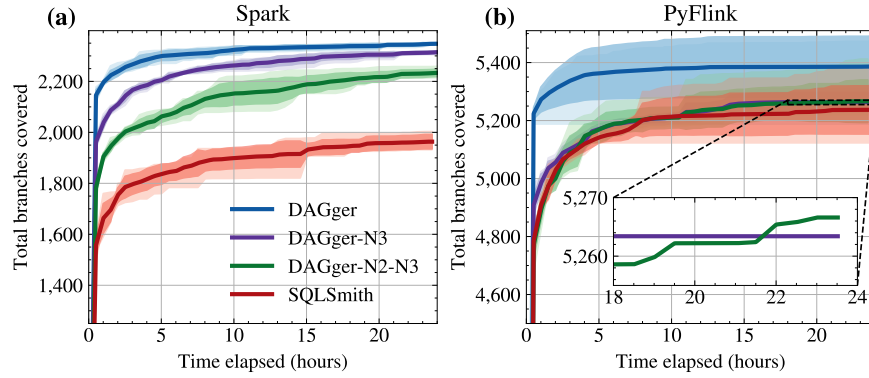


Figure 5.7: Branch Coverage for five trials of 24-hour fuzzing campaigns on Spark and PyFlink.

In frameworks where the baseline’s line coverage approaches **DAGGER**’s after 24 hours, we further assess the behavioral impact of the remaining uncovered lines (i.e., covered by **DAGGER** but not by the baseline) by measuring branch coverage. Figure 5.7 captures the branch coverage results from both **DAGGER** and baseline for Spark and Flink. As Figure 5.7 (a) shows, **DAGGER** achieves substantially higher final branch coverage (an increase of 16.3% in Spark branch coverage), indicating that while both tools reach similar code regions, **DAGGER** explores deeper behaviors within those regions. In other words, **DAGGER** more thoroughly exercises the control flow and code paths within the optimizer rules it reaches, uncovering behavioral diversity that SQLSmith fails to trigger. As expected, **DAGGER-N3** is able to explore more branches than **DAGGER-N2N3** since it is able to construct more valid dataflow graphs. Interestingly, while Figures 5.7 (b) shows a similar trend for **DAGGER** on PyFlink,

Table 5.1: Top component-level percentage line coverage differences between **DAGGER** and **SQLSmith** for each framework.

Frame- work	Category	Class / File	SQLSmith Cov.	DAGGER Cov.
Spark	DAGGER top advantage	<code>spark.sql.catalyst.expressions.ScalaUDF</code>	0.0% (0.0/9745.0)	12.2% (1184.4/9745.0)
	SQLSmith top advantage	<code>spark.sql.catalyst.parser.SqlBaseParser</code>	23.8% (9531.0/40117.0)	15.4% (6164.0/40117.0)
PyFlink	DAGGER top advantage	<code>flink.table.planner.plan.metadata. AggCallSelectivityEstimator</code>	13.7% (312.2/2286.0)	66.4% (1519.0/2286.0)
	SQLSmith top advantage	<code>flink.table.planner.plan.rules. SubQueryDecorrelator.SubQuery</code>	60.1% (643.6/1070.0)	0.0% (0.0/1070.0)
Polars	DAGGER top advantage	<code>polars-plan/src/plans/optimizer/predicate_ pushdown/join.rs</code>	0.0% (0/745)	17.1% (128/745)
	SQLSmith top advantage	<code>polars-plan/src/plans/optimizer/cse/csee.rs</code>	100.0% (433/590)	29.0% (171/590)
Dask	DAGGER top advantage	<code>dask/dataframe/dask_expr/_shuffle.py</code>	28.7% (162/565)	69.7% (394/565)
	SQLSmith top advantage	<code>dask/dataframe/dask_expr/_collection.py</code>	36.1% (735/2035)	33.3% (678/2035)

the two ablations perform similarly to each other. This is likely due to several factors such as an increased generation throughput and the two ablations exploring complementary branches. However, all the coverage graphs illustrate how combining both stages 2 and 3 in Section 5.3 is crucial for unlocking more coverage.

Overall, these results demonstrate that **DAGGER** not only improves total coverage but does so more consistently, efficiently, and deeply, owing to its higher validity rate and semantic-aware test generation approach.

Per Package Coverage Analysis

We analyze optimizer coverage by subpackage to identify where **DAGGER** provides the greatest advantage.

Table 5.1 illustrates systematic differences in components that are exercised by each approach.

On Spark, **DAGGER**’s largest coverage advantage appears in *ScalaUDF*. This aligns with the intended design of **DAGGER**’s that explicitly support user-defined functions. As a result, **DAGGER** triggers optimizer paths related to UDF expression handling, type binding, and expression rewriting. In contrast, SQLSmith’s strongest advantage is concentrated in the SQL frontend, most notably *SqlBaseParser*, which is expected given that SQLSmith operates exclusively by generating SQL strings and therefore heavily exercises parsing and AST construction.

PyFlink exhibits a similar overall pattern. SQLSmith’s coverage is dominated by the SQL parser `FlinkSqlParserImpl` class. One notable exception is query decorrelation, which, to the best of our knowledge, is among the few optimization behaviors expressed primarily through declarative SQL rather than through the DataFrame API, explaining SQLSmith’s relative advantage in decorrelation-related planner rules. **DAGGER**’s largest advantage appears in *AggCallSelectivityEstimator*. **DAGGER**’s DataFrame-based DAGs systematically produce richer aggregate-heavy plans, including aggregates composed with filters, joins, and nested operators. These plan shapes repeatedly invoke planner metadata routines responsible for estimating selectivity and cardinality for aggregate calls, which are used for cost-based optimization rule application. Therefore, **DAGGER** drives substantially higher coverage in selectivity estimation logic than SQL workloads.

Dask shows an analogous but framework-specific pattern. Within `dask/dataframe/dask_expr`, **DAGGER**’s strongest advantage is in `_shuffle.py`. This module is central to partition-rearrangement planning, including shuffle simplification and lowering into concrete execution strategies. **DAGGER**’s DAG-based workloads more frequently compose joins, group-bys, and repartition-like transformations, thereby exercising these shuffle-planning paths at a substantially higher rate than SQL-centric baselines. In particular, repartitioning abilities are not part of the standard SQL syntax and Dask does not support it via the SQL front

end. By contrast, SQLSmith’s top advantage appears in `_collection.py`, a broad collection API wrapper layer that mediates dataframe method dispatch and alignment behavior. This suggests SQLSmith’s relative gain is concentrated more in front-end paths, whereas DAGGER’s gain is concentrated in deeper execution-relevant planning logic.

Polars exhibits a complementary pattern within `polars-plan/src/plans/optimizer`. DAGGER’s largest advantage is in `predicate_pushdown/join.rs`: predicate pushdown moves filters earlier in the query plan to reduce data carried through joins, requiring join-aware rules to respect outer-join semantics. Since DAGGER’s DAG workloads routinely combine joins with filters, this logic is exercised heavily. SQLSmith’s Polars queries rarely produce successful plans with joins, leaving join-adjacent pushdown largely uncovered. SQLSmith’s strongest relative coverage is in `cse/csee.rs`, where Polars eliminates redundant subcomputations across projection columns; grammar-driven `SELECT` lists naturally exhibit such repetition, whereas DAGGER’s broader operator mix spends less mass in this projection-centric pattern. Overall, DAGGER’s advantage aligns with structural complexity, while SQLSmith’s aligns with repetitive select-and-project workloads that survive the SQL parser.

5.4.3 Bugs and Semantic Depth (RQ3)

In addition to consistently achieving high code coverage across all four frameworks, DAGGER demonstrates its effectiveness by identifying previously undiscovered optimizer bugs through its generated test DISC workloads. During the 24-hour campaigns, DAGGER uncovered a total of 9 new optimizer bugs: 4 in Flink, 2 in Spark, 2 in Polars, and 1 in Dask. These bugs stem from 30 unique outcome types, such as distinct exceptions, success indicators, or timeouts, observed across the evaluated frameworks’ unique error patterns, which are categorized based on error messages and the locations of errors in the framework code observed during the

```

1 source_table = table_env.from_elements(
2     data,
3     schema=schema
4 )
5 source_table.order_by(col('s_manager'))
6     .group_by(col('s_street_name'))
7     .select(col('s_gmt_offset'))
8     .count.alias('s_gmt_offset'))
9     .explain()

```

(a) A PyFlink program where the interplay of sorting and grouped aggregation triggers a crash.

```

1 def udf_agg(values: pd.Series):
2     return values.product()
3
4 source_table
5     .group_by(col('amt'))
6     .select(col('amt').avg.alias('amt'))
7     .distinct()
8     .group_by(col('amt'))
9     .select(col('amt').max.alias('amt'))
10    .group_by(col('amt'))
11    .select(call('udf_agg', col('amt')).alias('amt'))
12    .explain()

```

(b) A PyFlink program chaining repeated group-by and UDF aggregation stages that triggers a crash.

Figure 5.8: Examples of bug-inducing workloads generated by **DAGGER**. These programs expand into multi-stage execution DAGs involving repeated grouping, re-aggregation, operator reordering, and UDF boundaries. Although minimized for clarity, they induce deep planner pipelines with non-trivial operator dependencies that cannot be expressed as a single declarative SQL query, placing them outside the reach of SQL-only fuzzers such as SQLSmith.

testing campaigns. From these patterns, we selected and minimized a small sample of bugs before reporting them, in accordance with the issue reporting restrictions. Table 5.2 provides detailed information on the DISC framework bugs discovered by **DAGGER**.

To investigate further, we select two previously unreported bugs in PyFlink and present case studies that illustrate how **DAGGER**’s unique features contributed to identifying real-world bugs.

Case Study 1. Figure 5.8 illustrates two minimized PyFlink applications that trigger planner-level crashes during query plan generation. Figure 5.8(a) presents a PyFlink program

Table 5.2: Previously unknown bugs in DISC frameworks discovered by DAGGER in 24-hour test generation campaigns.

Frame-work	Github/Jira Issue	Bug Description (Simplified)	Bug Location	Impact
Dask	dask#12257	Incorrect division for partitioning leading index out of bounds exception.	Data Partitioning Logic	Crash
Flink	FLINK-38397	Stale sort trait causes BatchPhysicalSort.explainTerms to read non-existent index	Planner	Crash
Flink	FLINK-38366	Unable to generate execution plan when sort+aggregation on string column without a limit.	Optimizer	Crash
Flink	FLINK-38446	Optimizer crashes when using a custom UDAF while scheduling Hash Partitioning.	Planner	Crash
Flink	FLINK-38637	Optimizer crashed when orderBy was followed by limit.	Optimizer	Crash
Polars	polars#25971	Segmentation fault when right joining dataframes after sort and group_by	Join Optimizer	Crash
Polars	polars#26322	Optimizer panics while pushing down filter after right join in lazy mode.	Optimizer	Crash
Spark	SPARK-51798	Non-deterministic UDF property not respected when used in orderBy.	Optimizer	Wrong output
Spark	SPARK-54196	Phase planning fails with INTERNAL_ERROR when processing a query with case class objects.	Optimizer	Crash

that first sorts a source table by the column `s_manager`, then performs a `group_by` aggregation on `s_street_name`, and finally projects the count of `s_gmt_offset`. This sequence causes a mismatch between the planner’s *sort trait metadata* and the *actual schema* after projection. Specifically, the `order_by` operation retains all columns from the source table, but when the subsequent aggregation introduces a projection that narrows the schema, the sort trait continues to reference the original field positions. As a result, when `BatchPhysicalSort.explainTerms()` is invoked during plan generation, it attempts to access invalid field indices and throws an `ArrayIndexOutOfBoundsException`. This bug manifests only when the `sort_by` operation involves a column whose original index exceeds the number of columns remaining after aggregation and projection. This is a prime example of how DAGGER’s *state-view*-based DFG parametrization assisted in generating a valid operator parameter usage that causes unexpected planner behavior.

Case Study 2. Figure 5.8(b) shows another instance of a bug-revealing pyFlink application generated by DAGGER. It applies several `group_by` and `select` operations interleaved with a `distinct` call before applying a user-defined aggregation function (`udf_agg`). The PyFlink planner fails while generating a `BatchExecExchange` node, which represents a data shuffle between operators. Specifically, the `createHashPartition()` method raises an *IndexOutOfBoundsException*. This error is contingent on the presence of the custom aggregation UDF: replacing `udf_agg` with a built-in aggregation function prevents the crash. This bug scenario highlights how DAGGER discovered a planner bug owing to its ability to safely integrate arbitrary UDF-based aggregation in DAG, which enables UDF-based aggregation handling in the batch planner’s shuffle logic.

False Negative Analysis. While DAGGER has an oracle to flag incorrect UDF duplication, the current prototype is mostly focused on detecting crashes rather than correctness issues. Since DAGGER does not analyze the output of the DISC program, it will not classify as bugs the programs that have faulty outputs unless they also trigger a crash or contain wrongly duplicated UDFs. While in theory, the user can easily use DAGGER to run the program and compare outputs, we do not do this in our experiments since, unlike DBMS queries, each run requires running in a distributed fashion, which is expensive and time-consuming. In fact, even SQLancer struggles to execute more than one query per second when run on Apache Spark.

False Positive Analysis. DAGGER’s oracles detect a mismatch between optimized and unoptimized versions. Since we compare exit status differences, there should be no difference between the optimized and unoptimized versions in terms of compilation success. Therefore, the false-positive rate of this crash test is zero.

5.4.4 Generalizability (RQ4)

We evaluate the generalizability of **DAGGER** with respect to its ability to target different dataflow frameworks, focusing on the implementation effort required to support a new framework. Table 5.3 reports the LOC contributed by one of the authors when adding support for each framework discussed in this chapter. While LOC alone cannot fully capture development complexity, it provides a useful approximation of engineering effort. Additionally, we document and report the time effort (in Table 5.3) one of the authors put into implementing support for the new DISC frameworks.

Table 5.3: Lines of code comparison across frameworks for Dagger vs SQLSmith. **codegen** covers operator and query generation logic (C++ for SQLSmith, Scala for DAGger). **environment** covers runtime configuration, test launching, and oracle implementation; SQL-Smith requires additional binding code since it lacks native DISC framework support.

Framework	#LOC codegen		#LOC environment	
	DAGGER	SQLsmith	DAGGER	SQLsmith
Spark	558	2357	241	762
Flink	611	2357	1143	1668
Dask	626	2357	1042	1632
Polars	541	2357	1055	1667

For instance, **DAGGER**'s support for Dask [91], an open-source Python library for parallel computing, is implemented in under 626 LOC for Dask code generation logic and 1055 LOC for launching tests, runtime setup, execution harnesses, and implementing the oracle. It took the author approximately 3 hours to complete this implementation, which also involved copy-pasting from existing support for PyFlink due to overlap in the API specifications. Note that the author had no prior familiarity with or experience with Dask or its API. On average, only 1,454 lines of code were required to integrate a new framework into **DAGGER**. In comparison, fully adapting the baseline, SQLSmith, to the dataframe API requires modification in nearly 3,789 lines of SQLSmith code, and even then, it will still lack support for state-view-based parameterization. Similarly, adding support for a new SQL dialect to SQL fuzzer like

SQLancer [18] typically requires thousands of lines of Java code, with certain extensions (e.g., `sqlite3`) exceeding 9,000 lines.

5.5 Chapter Summary

Despite the widespread use of dataflow APIs in DISC systems like Apache Spark, Flink, and Polars, no general-purpose workload generation framework exists for testing them. Existing approaches, especially SQL-based fuzzers, are tied to specific syntactic front-ends and offer limited control over underlying dataflow structure and semantics. We presented **DAGGER**, a DAG-based fuzzer that generates semantically valid dataflow programs by synthesizing abstract operator graphs and lowering them into executable queries. By treating the logical operator DAG as the core unit of workload generation, **DAGGER** naturally captures data dependencies, operator composition, and schema evolution in a way that mirrors modern dataflow execution models. Evaluation across Apache Spark, PyFlink, Dask, and Polars shows that **DAGGER** achieves higher optimizer reachability (22.7% vs. 2.9% for SQLSmith) and 15.5% greater line coverage, uncovering 9 bugs within 24 hours. Its modular, engine-agnostic architecture enables straightforward extension to other dataflow systems, opening new avenues for systematic testing across distributed platforms.

Chapter 6

Future Directions

This thesis shows that using fine-grained workload information, both from data and program structure, can significantly enhance automated testing throughout the DISC stack. The three contributions discussed (*i.e.* **DEPFUZZ**, **NATURALFUZZ**, and **DAGGER**) each focus on a specific aspect of the DISC workflow. However, there are still several additional areas within DISC that require further investigation.

6.1 Testing Streaming Applications

A central capability of modern DISC frameworks is support for unbounded, event-driven computation. Systems such as Apache Spark Structured Streaming and Apache Flink expose streaming APIs in which programs consume continuously arriving records, apply relational operators over logical streams, and emit incremental results under event-time semantics [14]. A representative pattern is a tumbling-window aggregation with watermarks and allowed lateness, e.g., `readStream(...).withWatermark(...).groupBy(window(col("ts"), "10 minutes"), key).count()`, where correctness depends on ingestion order, window boundaries, and trigger policies rather than on a single finite input batch.

This thesis is the first to present dedicated techniques for testing DISC framework optimizers through workload generation, but the techniques assume batch processing. Many production frameworks maintain distinct batch and streaming planner and runtime subsystems; streaming-

specific workloads and rewrite rules (e.g., watermark propagation) therefore remain outside the scope of the work. **DAGGER** may still surface planning-time faults for some streaming constructs, but more work needs to be done to address challenges unique to streaming workloads.

Extending concepts in this thesis to streaming raises fundamental test-oracle challenges that do not arise in batch mode. First, outputs are unbounded or arrival-dependent, so there is no fixed reference relation to compare at job completion. Second, observable results depend on event-time assignment, watermark advancement, and handling of late records, making identical logical programs potentially produce different results. Third, stateful operators couple correctness to internal operator state and retention policies, which are difficult to reproduce without controlling the stream source. Fourth, delivery semantics (at-most-once, at-least-once, exactly-once) and integration with external sinks introduce timing-sensitive side effects that violate the deterministic replay assumptions underlying plan-based comparison. Addressing these issues will require controlled stream generation and time-aware oracles tailored to streaming workloads.

6.2 Rule-Targeted Testing

While this thesis introduces a new correctness oracle, i.e. the plan-comparison oracle that detects violations of non-deterministic UDF contracts by comparing optimized and unoptimized logical plans, there is room for more research on dedicated correctness oracles.

Because **DAGGER** is designed as a general-purpose workload generator, its search strategy prioritizes broad operator diversity and optimizer reachability rather than steering mutations toward individual rewrite rules. Consequently, most defects reported to date manifest as exceptions or crashes during planning or execution. A complementary research direction is

rule-targeted test generation: characterize the preconditions and effects of specific optimizer rules (e.g., predicate pushdown, column pruning, or sort elimination), then synthesize workloads that are structurally likely to trigger those rules while supplying differential oracles tailored to their semantics. This could improve both the rate of semantic fault detection and the diagnosability of optimizer regressions.

Realizing rule-targeted generation at scale faces several challenges. Optimizer rule sets are large, version-dependent, and often interact, so a workload intended to exercise one rewrite may be transformed by others first or never reach the target rule. Precise preconditions are rarely documented as executable specifications, which makes it difficult to synthesize programs that reliably *fire* a rule without. Even when a rule is applied, establishing correctness requires oracles that can detect it e.g. a buggy filter-pushdown might not be detected if the data is not tailored. A holistic and effective solution would likely require some coordination between data generation and program generation.

6.3 Cross-Framework Differential Testing

A defining characteristic of the modern DISC landscape is the proliferation of competing frameworks that expose semantically equivalent programming models over distinct execution engines. Systems such as Apache Spark, Apache Flink, and Dask each provide dataframe and relational APIs through which users express transformations such as joins, aggregations, and window functions, yet they may differ substantially in their optimizer architectures, physical execution strategies, and handling of edge cases. A representative example is a grouped aggregation with a filter predicate, e.g., `df.filter(col("v") > 0).groupBy("k").agg(sum("v"))`, which may produce identical results across frameworks on well-formed inputs but diverge on inputs containing nulls, duplicates, or skewed key distributions due to differences in optimizer rewrite

rules and runtime evaluation order.

The workload generation techniques developed in this thesis target individual frameworks in isolation. Consequently, semantic disagreements that arise from framework-level differences in query semantics remain outside the fault model considered here.

Extending the workload generation approach to cross-framework differential testing introduces several open challenges. First, establishing semantic equivalence between programs written against different APIs requires a mapping layer that preserves operator intent across syntactic and behavioral variations; operators that appear equivalent (e.g., `join` in Spark versus Flink) may differ in their treatment of duplicate keys or time-based semantics. Second, not all frameworks expose the same operator surface, so workloads must be restricted to a portable subset while remaining expressive enough to exercise optimizer logic. Third, differences in output may reflect legitimate behavioral variation rather than bugs, requiring oracles that can distinguish specification-level divergence from implementation faults. Such an approach could surface latent inconsistencies in how frameworks implement the relational algebra, complementing the single-framework fault detection demonstrated in this thesis.

Chapter 7

Conclusion

This thesis presents a principled approach to fuzzing Data-Intensive Scalable Computing (DISC) systems by leveraging structural and semantic workload information, across both input data and program logic, to improve the effectiveness of automated testing. Existing fuzzers fall short in this domain due to the scale and complexity of inputs, the need for coordinated mutations across datasets, and the opaque behavior of user-defined functions embedded in dataflow graphs.

The first part of this work focuses on enhancing input mutation for DISC applications by analyzing how programs interact with data at the row and column level. This enables the generation of targeted, constraint-aware test inputs that reach deeper logic in applications, beyond superficial parsing errors. The second part addresses the challenge of input realism by introducing strategies to generate natural and semantically meaningful test data. These techniques maintain the structural integrity of real-world datasets while improving coverage and fault detection, making test cases more useful for developers. The final part extends fuzzing to the frameworks themselves, proposing a DAG-based approach for generating diverse dataflow programs that systematically exercise internal components such as query optimizers and execution engines.

Collectively, these contributions demonstrate that effective testing of DISC systems requires more than surface-level input mutation. By understanding how data and computation interact, this work advances the design of fuzzers that are both powerful and practical, capable of

uncovering deep faults, producing interpretable test inputs, and scaling to the complexity of modern dataflow-based systems.

Bibliography

- [1] Depfuzz. <https://github.com/SEED-VT/DepFuzz>. Accessed: 2023-01-10.
- [2] American fuzzy lop. <https://lcamtuf.coredump.cx/afl/>, 2021. Accessed: 2021-12-14.
- [3] Apache hadoop. <https://hadoop.apache.org/>, 2022. Accessed: 2021-12-14.
- [4] Apache spark. <https://spark.apache.org/>, 2022. Accessed: 2021-12-14.
- [5] Migration guide: Sql, datasets and dataframe — spark documentation. <https://spark.apache.org/docs/latest/sql-migration-guide.html>, 2025. Accessed: 2025-10-11.
- [6] Big data benchmark, Accessed: 2022-09-01. URL <https://amplab.cs.berkeley.edu/benchmark/>.
- [7] Tpc-ds version 2 and version 3, Accessed: 2022-09-01. URL <https://www.tpc.org/tpcds/default5.asp>.
- [8] Welcome to faker’s documentation!, Accessed: 2023. URL <https://faker.readthedocs.io/en/master/>.
- [9] Demonstration database. <https://postgrespro.com/community/demodb>, Accessed: 2023-01-10.
- [10] Spark programming guide, Accessed: 2023-05-04. URL <https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#accumulators>.
- [11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur,

- Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi>.
- [12] Abdullah Alsharif, Gregory M. Kapfhammer, and Phil McMinn. Domino: Fast and effective test data generation for relational database schemas. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 12–22, 2018. doi: 10.1109/ICST.2018.00012.
- [13] Anthropic. Build with claude, 2024. URL <https://www.anthropic.com/learn/build-with-claude>. Accessed: 2025-01-30.
- [14] Apache Software Foundation. Apache spark: Unified engine for large-scale data analytics. <https://spark.apache.org/>, 2014. Open-source cluster computing framework for large-scale data processing.
- [15] Apache Spark. Versioning policy. <https://spark.apache.org/versioning-policy.html>, 2026. Accessed: 2026-01-30.
- [16] Apache Spark Documentation. pyspark.sql.functions.udf — pyspark api reference. <https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.udf.html>, 2026. Accessed 30 Jan 2026.
- [17] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *ICSE 2019*, November 2019. URL <https://www.microsoft.com/en-us/research/publication/restler-stateful-rest-api-fuzzing/>.

- [18] Jinsheng Ba and Manuel Rigger. Testing database engines via query plan guidance. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2060–2071, 2023. doi: 10.1109/ICSE48619.2023.00174.
- [19] Jinsheng Ba and Manuel Rigger. Cert: Finding performance issues in database systems through the lens of cardinality estimation. In *Proceedings of the 46th International Conference on Software Engineering (ICSE 2024)*, Lisbon, Portugal, 2024. ISBN 979-8-4007-0217-4/24/04. doi: 10.1145/3597503.3639076.
- [20] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. A genetic approach for random testing of database systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB '07*, page 1243–1251. VLDB Endowment, 2007. ISBN 9781595936493.
- [21] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A taint based approach for smart fuzzing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 818–825, 2012. doi: 10.1109/ICST.2012.182.
- [22] Michael Beyene and James H. Andrews. Generating string test data for code coverage. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 270–279, 2012. doi: 10.1109/ICST.2012.107.
- [23] Alexander Boyce and Mathieu Leger. Stateful streaming with apache spark: How to update decision logic at runtime, Accessed: 2023-01-10. URL https://www.databricks.com/session_eu20/stateful-streaming-with-apache-spark-how-to-update-decision-logic-at-runtime.
- [24] Nicolas Bruno and Surajit Chaudhuri. Flexible database generators. In *Proceedings of the 31st International Conference on Very Large Data Bases, VLDB '05*, page 1097–1107. VLDB Endowment, 2005. ISBN 1595931546.

- [25] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2019. doi: 10.1109/TSE.2017.2785841.
- [26] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, and Volker Markl. Apache flink™: Stream and batch processing in a single engine. In *IEEE Data Engineering Bulletin*, volume 43, pages 28–38, 2020. URL <https://flink.apache.org>. PyFlink provides Python APIs for Apache Flink, enabling unified stream and batch data processing in Python.
- [27] Deepayan Chakrabarti, Ravi Kumar, and Kunal Punera. A graph-theoretic approach to webpage segmentation. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, page 377–386, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580852. doi: 10.1145/1367497.1367549. URL <https://doi.org/10.1145/1367497.1367549>.
- [28] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 46(5):709–719, April 2015. doi: 10.1002/spe.2325. URL <https://doi.org/10.1002/spe.2325>.
- [29] Zaheer Chothia, John Liagouris, Frank McSherry, and Timothy Roscoe. Explaining outputs in modern data analytics. *Proc. VLDB Endow.*, 9(12):1137–1148, August 2016. ISSN 2150-8097. doi: 10.14778/2994509.2994530. URL <http://dx.doi.org/10.14778/2994509.2994530>.
- [30] David Coppit and Jiexin Lian. Yagg: An easy-to-use generator for structured test inputs. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, page 356–359, New York, NY, USA, 2005. Association

- for Computing Machinery. ISBN 1581139934. doi: 10.1145/1101908.1101969. URL <https://doi.org/10.1145/1101908.1101969>.
- [31] `dask-contrib`. `dask-sql` README. <https://github.com/dask-contrib/dask-sql/blob/775b56fb8f99a5e061bc9c9fc86a035554489578/README.md>, 2026. GitHub repository README, commit 775b56fb8f99a5e061bc9c9fc86a035554489578, accessed 2026-05-12.
- [32] Dask Developers. Does dask implement sql? — dask dataframe and sql. <https://docs.dask.org/en/latest/dataframe-sql.html#does-dask-implement-sql>, 2026. Accessed 30 Jan 2026.
- [33] Databricks. Introducing recursive common table expressions: Making databricks sql turing complete, July 2025. URL <https://www.databricks.com/blog/introducing-recursive-common-table-expressions-databricks>. Databricks Blog.
- [34] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 137–150. USENIX Association, 2004. URL <https://www.usenix.org/legacy/event/osdi04/tech/dean.html>. Introduces the MapReduce programming model for processing and generating large datasets on clusters.
- [35] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '23*, pages 423–435. ACM, 2023. doi: 10.1145/3597926.3598067.

- [36] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case fuzzers: Testing deep learning libraries via FuzzGPT. In *Proceedings of the 46th International Conference on Software Engineering*, ICSE '24, pages 1–13. ACM, 2024. doi: 10.1145/3597503.3623343.
- [37] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [38] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, 2016. doi: 10.1109/SP.2016.15.
- [39] EcEmma Team. Jacoco – java code coverage library. <https://www.jacoco.org/>, 2009. Version 0.8.x or later.
- [40] Fariza Fauzi, Jer-Lang Hong, and Mohammed Belkhatir. Webpage segmentation for extracting images and their surrounding contextual information. In *Proceedings of the 17th ACM International Conference on Multimedia*, MM '09, page 649–652, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605586083. doi: 10.1145/1631272.1631379. URL <https://doi.org/10.1145/1631272.1631379>.
- [41] Myles Foley and Sergio Maffeis. Apirl: Deep reinforcement learning for rest api fuzzing. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39:191–199, 04 2025. doi: 10.1609/aaai.v39i1.31995.
- [42] Ying Fu, Zhiyong Wu, Yuanliang Zhang, Jie Liang, Jingzhou Fu, Yu Jiang, Shanshan Li, and Xiangke Liao. Thanos: Dbms bug detection via storage engine rotation based

- differential testing. In *Proceedings of the 47th International Conference on Software Engineering (ICSE 2025)*. IEEE / ACM, 2025. doi: 10.1109/ICSE55347.2025.00001.
- [43] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484, 2009. doi: 10.1109/ICSE.2009.5070546.
- [44] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, jun 2008. ISSN 0362-1340. doi: 10.1145/1379022.1375607. URL <https://doi.org/10.1145/1379022.1375607>.
- [45] Anastasios Gounaris, Georgia Kougka, Ruben Tous, Carlos Tripiana Montes, and Jordi Torres. Dynamic configuration of partitioning in spark applications. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):1891–1904, 2017. doi: 10.1109/TPDS.2017.2647939.
- [46] Muhammad Ali Gulzar, Matteo Interlandi, Tyson Condie, and Miryung 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 1033–1037, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2983930. URL <https://doi.org/10.1145/2950290.2983930>.
- [47] Muhammad Ali Gulzar, Matteo Interlandi, Xueyuan Han, Mingda Li, Tyson Condie, and Miryung Kim. Automated debugging in data-intensive scalable computing. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 520–534, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350280. doi: 10.1145/3127479.3131624. URL <https://doi.org/10.1145/3127479.3131624>.

- [48] Muhammad Ali Gulzar, Shaghayegh Mardani, Madanlal Musuvathi, and Miryung 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 290–301, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. doi: 10.1145/3338906.3338953. URL <https://doi.org/10.1145/3338906.3338953>.
- [49] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi: 10.1145/3371080. URL <https://doi.org/10.1145/3371080>.
- [50] Kenneth Houkjær, Kristian Torp, and Rico Wind. Simple and realistic data generation. In *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB '06*, page 1243–1246. VLDB Endowment, 2006.
- [51] Ahmad Humayun, Miryung Kim, and Muhammad Ali Gulzar. Co-dependence aware fuzzing for dataflow-based big data analytics. ESEC/FSE 2023, page 1050–1061, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703270. doi: 10.1145/3611643.3616298. URL <https://doi.org/10.1145/3611643.3616298>.
- [52] Ahmad Humayun, Miryung Kim, and Muhammad Ali Gulzar. Depfuzz tool for ”co-dependence aware fuzzing for dataflow-based big data analytics”, 2023.
- [53] Ahmad Humayun, Yaoxuan Wu, Miryung Kim, and Muhammad Ali Gulzar. Naturalfuzz: Natural input generation for big data analytics. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering, ASE '23*,

- page 1592–1603. IEEE Press, 2024. ISBN 9798350329964. doi: 10.1109/ASE56229.2023.00034. URL <https://doi.org/10.1109/ASE56229.2023.00034>.
- [54] 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, April 2012. doi: 10.1109/ICDE.2012.118.
- [55] R. Ikeda, A. Das Sarma, and J. Widom. Logical provenance in data-oriented workflows? In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 877–888, April 2013. doi: 10.1109/ICDE.2013.6544882.
- [56] Robert Ikeda, Hyunjung Park, and Jennifer Widom. Provenance for generalized map and reduce workflows. In *In Proc. Conference on Innovative Data Systems Research (CIDR)*, 2011.
- [57] Code Intelligence. Jazzer. <https://github.com/CodeIntelligenceTesting/jazzer>, 2022.
- [58] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. Titian: Data provenance support in spark. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, volume 9, page 216. NIH Public Access, 2015. doi: 10.1007/s00778-017-0474-5.
- [59] jamesshocking. Spark-rest-api-udf-scala: How to execute a rest api call in apache spark the right way, using scala. <https://github.com/jamesshocking/Spark-REST-API-UDF-Scala>, 2026. GitHub repository, accessed 30 Jan 2026.

- [60] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.
- [61] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. DynSQL: Stateful fuzzing for database management systems with complex and valid SQL query generation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4949–4965, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL <https://www.usenix.org/conference/usenixsecurity23/presentation/jiang-zu-ming>.
- [62] René Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 433–436, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. doi: 10.1145/2610384.2628053. URL <https://doi.org/10.1145/2610384.2628053>.
- [63] Kavi, Buckles, and Bhat. A formal definition of data flow graph models. *IEEE Transactions on Computers*, C-35(11):940–948, 1986. doi: 10.1109/TC.1986.1676696.
- [64] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE ’18, page 475–485, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375. doi: 10.1145/3238147.3238176. URL <https://doi.org/10.1145/3238147.3238176>.
- [65] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE ’23, pages 919–931. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00085.

- [66] Kaituo Li, Christoph Reichenbach, Yannis Smaragdakis, Yanlei Diao, and Christoph Csallner. Sedge: Symbolic example data generation for dataflow programs. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 235–245, 2013. doi: 10.1109/ASE.2013.6693083.
- [67] Nan Li, Yu Lei, Haider Riaz Khan, Jingshu Liu, and Yun Guo. Applying combinatorial test data generation to big data applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE '16*, page 637–647, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338455. doi: 10.1145/2970276.2970325. URL <https://doi.org/10.1145/2970276.2970325>.
- [68] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017*, page 627–637, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. doi: 10.1145/3106237.3106295. URL <https://doi.org/10.1145/3106237.3106295>.
- [69] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun. Pata: Fuzzing with path aware taint analysis. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 154–170, Los Alamitos, CA, USA, may 2022. IEEE Computer Society. doi: 10.1109/SP46214.2022.9833594. URL <https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00010>.
- [70] Dongge Liu, Oliver Chang, Jonathan Metzman, Martin Sablotny, and Mihai Maruseac. OSS-Fuzz-Gen: Automated fuzz target generation. <https://github.com/google/oss-fuzz-gen>, 2024.
- [71] LLVM Project. llvm-cov — emit coverage information — llvm command guide. <https://llvm.org/docs/CommandGuide/llvm-cov.html>, 2026. Accessed 30 Jan 2026.

- [72] Dionysios Logothetis, Soumyarupa De, and Kenneth 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. ISBN 9781450324281. doi: 10.1145/2523616.2523619. URL <https://doi.org/10.1145/2523616.2523619>.
- [73] Chenyang Lyu, Jiacheng Xu, Shouling Ji, Xuhong Zhang, Qinying Wang, Binbin Zhao, Gaoning Pan, Wei Cao, Peng Chen, and Raheem Beyah. MINER: A hybrid Data-Driven approach for REST API fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4517–4534, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL <https://www.usenix.org/conference/usenixsecurity23/presentation/lyu>.
- [74] maropu. fuzz-testing-for-spark: Sql-aware fuzz tests for the apache spark catalyst optimizer. <https://github.com/maropu/fuzz-testing-for-spark>, 2026. GitHub repository, accessed 30 Jan 2026.
- [75] P.M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, 1990. doi: 10.1109/52.56422.
- [76] Ehsan Mazloumi, Graham Currie, and Geoffrey Rose. Using gps data to gain insight into public transport travel time variability. *Journal of Transportation Engineering*, 136(7):623–631, 2010. doi: 10.1061/(ASCE)TE.1943-5436.0000126. URL <https://ascelibrary.org/doi/abs/10.1061/%28ASCE%29TE.1943-5436.0000126>.
- [77] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium*, NDSS '24. Internet Society, 2024. doi: 10.14722/ndss.2024.24556.

- [78] Farhad Nabhani and Alireza Shokri. Reducing the delivery lead time in a food distribution SME through the implementation of six sigma methodology. *Journal of Manufacturing Technology Management*, 20(7):957–974, September 2009. doi: 10.1108/17410380910984221. URL <https://doi.org/10.1108/17410380910984221>.
- [79] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *VLDB*, volume 6, pages 1049–1058, 2006.
- [80] Ned Batchelder and Coverage.py Developers. Coverage.py documentation (version 7.13.2). <https://coverage.readthedocs.io/en/7.13.2/>, 2026. Accessed 30 Jan 2026.
- [81] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. Generating example data for dataflow programs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 245–256, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585512. doi: 10.1145/1559845.1559873. URL <https://doi.org/10.1145/1559845.1559873>.
- [82] Alessandro Orso and Gregg Rothermel. Software testing: A research travelogue (2000–2014). In *Future of Software Engineering Proceedings*, FOSE 2014, page 117–132, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328654. doi: 10.1145/2593882.2593885. URL <https://doi.org/10.1145/2593882.2593885>.
- [83] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 329–340, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362245. doi: 10.1145/3293882.3330576. URL <https://doi.org/10.1145/3293882.3330576>.

- [84] Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. The synthetic data vault. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 399–410, 2016. doi: 10.1109/DSAA.2016.49.
- [85] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710, 2018. doi: 10.1109/SP.2018.00056.
- [86] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
- [87] Nihar Ranjan, Kaushal Mundada, Kunal Phaltane, and Saim Ahmad. A survey on techniques in nlp. *International Journal of Computer Applications*, 134(8):6–9, 2016.
- [88] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi: 10.1145/3428279. URL <https://doi.org/10.1145/3428279>.
- [89] Mikael Rittri. Using types as search keys in function libraries. In *Proceedings of the fourth international conference on Functional Programming languages and Computer Architecture*, pages 174–183, 1989.
- [90] Roch, Grzegorz Slowikowski, Roland Tritsch, Sam, and Chris Kipp. scoverage. <https://github.com/scoverage>, 2022. Accessed: 2022-01-10.
- [91] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference (SciPy 2015)*, pages 130–136, 2015.
- [92] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. Bug synthesis: Challenging bug-finding tools with deep faults. 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 224–234, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. doi: 10.1145/3236024.3236084. URL <https://doi.org/10.1145/3236024.3236084>.
- [93] Andreas Seltenreich. Sqlsmith: Randomized sql query generator. <https://github.com/anse1/sqlsmith>, 2015. Accessed: 2025-04-26.
- [94] Kostya Serebryany. Libfuzzer – a library for coverage-guided fuzz testing., Accessed: 2023-01-29. URL <https://llvm.org/docs/LibFuzzer.html>.
- [95] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. Neutaint: Efficient dynamic taint analysis with neural networks. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1527–1543, 2020. doi: 10.1109/SP40000.2020.00022.
- [96] Tatsuya Shima, Etienne Bacher, and Authors of the dependency Rust crates. *polars: R Bindings for the 'polars' Rust Library*, 2026. URL <https://github.com/pola-rs/r-polars>. R package version 1.7.0.9000.
- [97] Emin Gün Sirer and Brian N. Bershad. Using production grammars in software testing. *SIGPLAN Not.*, 35(1):1–13, dec 2000. ISSN 0362-1340. doi: 10.1145/331963.331965. URL <https://doi.org/10.1145/331963.331965>.
- [98] SparkByExamples.com. Spark map() vs mappartitions() transformation with examples. <https://sparkbyexamples.com/spark/spark-map-vs-mappartitions-transformation/>, 2025. Accessed 30 Jan 2026.
- [99] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller:

- Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.
- [100] Jason Teoh, Muhammad Ali Gulzar, and Miryung Kim. Influence-based provenance for dataflow applications with taint propagation. In *Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20*, page 372–386, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421292. URL <https://doi.org/10.1145/3419111.3421292>.
- [101] Alexandre Vianna, Waldemar Ferreira, and Kiev Gama. An Exploratory Study of How Specialists Deal with Testing in Data Stream Processing Applications . In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, Los Alamitos, CA, USA, September 2019. IEEE Computer Society. doi: 10.1109/ESEM.2019.8870186. URL <https://doi.ieeecomputersociety.org/10.1109/ESEM.2019.8870186>.
- [102] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512, 2010. doi: 10.1109/SP.2010.37.
- [103] Yuchen Wang, Tianyi Chen, Yingnong Xiong, Chenguang Zhang, Zhaoguo Xu, Dan Meng, Yu Zhou, Xiaojun Xu, Shimin Zhang, Xiaoxing Ma, and Lu Zhang. Puppy: Finding performance degradation bugs in dbmss via limited-optimization plan construction. In *Proceedings of the 47th International Conference on Software Engineering (ICSE 2025)*. ACM, 2025. doi: 10.1145/3660805.3667828.
- [104] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the*

- Foundations of Software Engineering*, ESEC/FSE 2020, page 788–799, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409761. URL <https://doi.org/10.1145/3368089.3409761>.
- [105] Yaoxuan Wu, Ahmad Humayun, Muhammad Ali Gulzar, and Miryung Kim. Natural symbolic execution-based testing for big data analytics. *Proc. ACM Softw. Eng.*, 1 (FSE), July 2024. doi: 10.1145/3660825. URL <https://doi.org/10.1145/3660825>.
- [106] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4All: Universal fuzzing with large language models. In *Proceedings of the 46th International Conference on Software Engineering*, ICSE ’24, pages 126:1–126:13. ACM, 2024. doi: 10.1145/3597503.3639121.
- [107] Guoqing Harry Xu, Margus Veanes, Michael Barnett, Madan Musuvathi, Todd Mytkowicz, Ben Zorn, Huan He, and Haibo Lin. Nijima: Sound and automated computation consolidation for efficient multilingual data-parallel pipelines. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP ’19, page 306–321, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359649. URL <https://doi.org/10.1145/3341301.3359649>.
- [108] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. Modeling tabular data using conditional gan. *Advances in Neural Information Processing Systems*, 32, 2019.
- [109] Chengxu Yang, Yuanchun Li, Mengwei Xu, Zhenpeng Chen, Yunxin Liu, Gang Huang, and Xuanzhe Liu. Taintstream: Fine-grained taint tracking for big data platforms through dynamic code translation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 806–817, New York, NY, USA, 2021.

- Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3468532. URL <https://doi.org/10.1145/3468264.3468532>.
- [110] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion 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*, NSDI'12, page 2, USA, 2012. USENIX Association.
- [111] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 722–733, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450367684. doi: 10.1145/3324884.3416641. URL <https://doi.org/10.1145/3324884.3416641>.
- [112] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 955–970, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. doi: 10.1145/3372297.3417260. URL <https://doi.org/10.1145/3372297.3417260>.