# NaturalFuzz: Natural Input Generation for Big Data Analytics

Ahmad Humayun
*Virginia Tech*
Blacksburg, USA
ahmad35@vt.edu

Yaoxuan Wu
*UCLA*
Los Angeles, USA
thaddywu@cs.ucla.edu

Miryung Kim
*UCLA*
Los Angeles, USA
miryung@cs.ucla.edu

Muhammad Ali Gulzar
*Virginia Tech*
Blacksburg, USA
gulzar@cs.vt.edu

***Abstract*—Fuzzing applies input mutations iteratively with the only goal of finding more bugs, resulting in synthetic tests that tend to lack realism. Big data analytics are expected to ingest *real-world data* as input. Therefore, when synthetic test data are not easily comprehensible, they are less likely to facilitate the downstream task of fixing errors. Our position is that fuzzing in this domain must achieve both high naturalness and high code coverage. We propose a new *natural* synthetic test generation tool for big data analytics, called NATURALFUZZ. It generates both semi-structured and structured data with corresponding semantics such as 'zipcode' and 'age.' The key insights behind NATURALFUZZ are two-fold. First, though existing test data may be small and lack coverage, we can grow this data to increase code coverage. Second, we can strategically mix constituent parts across different rows and columns to construct new realistic synthetic data by leveraging fine-grained data provenance.**

**On commercial big data application benchmarks, NATURALFUZZ achieves an additional $19.9\%$ coverage and detects $1.9\times$ more faults than a machine learning-based synthetic data generator (SDV) when generating comparably sized inputs. This is because an ML-based synthetic data generator does not consider which code branches are exercised by which input rows from which tables, while NATURALFUZZ is able to select input rows that have a high potential to increase code coverage and mutate the selected data towards unseen, new program behavior. NATURALFUZZ's test data is more realistic than the test data generated by two baseline fuzzers (BigFuzz and Jazzer), while increasing code coverage and fault detection potential. NATURALFUZZ is the first fuzzing methodology with three benefits: (1) exclusively generate natural inputs, (2) fuzz multiple input sources simultaneously, and (3) find deeper semantics faults.**

## I. INTRODUCTION

Data-Intensive Scalable Computing (DISC) applications are becoming increasingly popular for processing large amounts of data. Frameworks like Hadoop MapReduce [1] and Apache Spark [2] provide APIs to the developers that allow them to manipulate the data at scale. These frameworks distribute the data and application on thousands of machines in a cluster so each machine can work on an independent chunk of data in parallel. Despite the widespread usage of such applications, automated testing remains a major challenge due to unstructured natural inputs coupled with the scale of the data and the application's distributed nature.

Fuzzing is a prevalent automated testing technique. It repetitively tests a program with randomly mutated data to expose software faults [3], [4], [5], [6], [7], [8], [9], [10], [11]. Nearly all fuzzing techniques 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) [13] 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 [14]. We compare our technique against two available baseline techniques: Jazzer [15], a coverage-guided greybox fuzzer for Java bytecode based on LibFuzzer [16]; and BIGFUZZ [4], 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× more faults than JAZZER and 1.7× 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* [17], 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 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

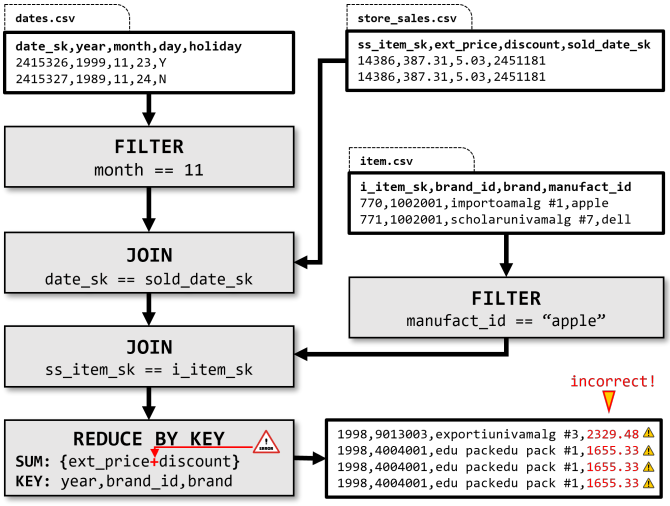


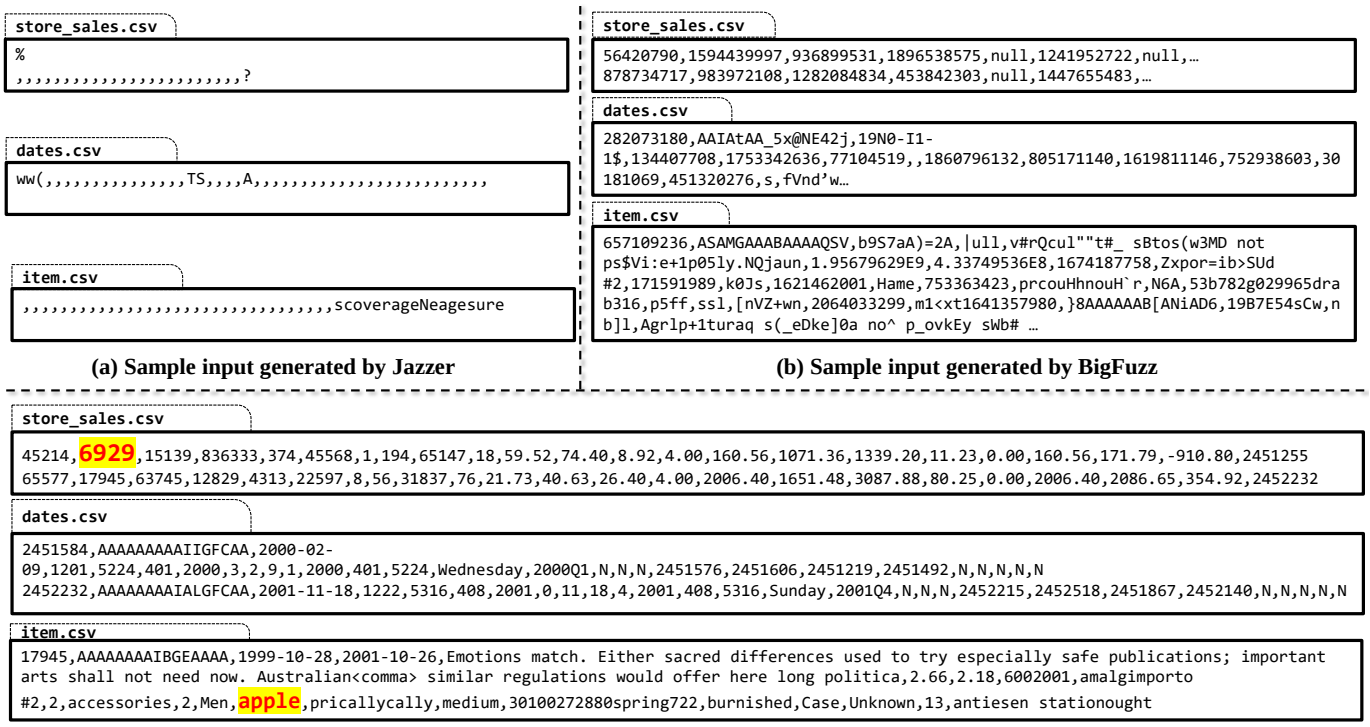

Fig. 1: Dataflow graph of a program that computes the total sales price of items from a particular manufacturer for each year. The + operator in the `reduceByKey` should be a − and is incorrect, leading to incorrect values in the output.

## II. 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 a particular manufacturer for a given store. Figure 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.



**(c) Input generated by NaturalFuzz that exposes fault. Mutated columns are highlighted in yellow.**

Fig. 2: Sample inputs generated by each tool.

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 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 1.

```
def test_oracle(output_row, original_price):
 if (output_row.discounted_price > original_price)
    false
 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 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 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 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.

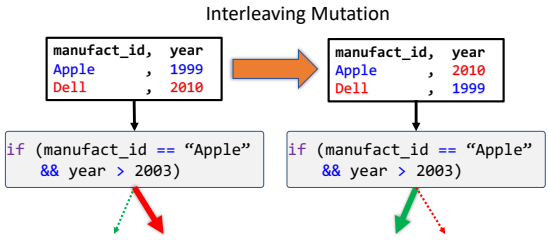



Fig. 3: How interleaving unlocks new paths in the program

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 1-(c), triggering the faulty code line. The outcomes generated using these inputs indeed include rows where test oracle is violated, *i.e.,* , a discounted price is greater than the original price.

## III. 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 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.

### A. 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 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.

### B. 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 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 [18] 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

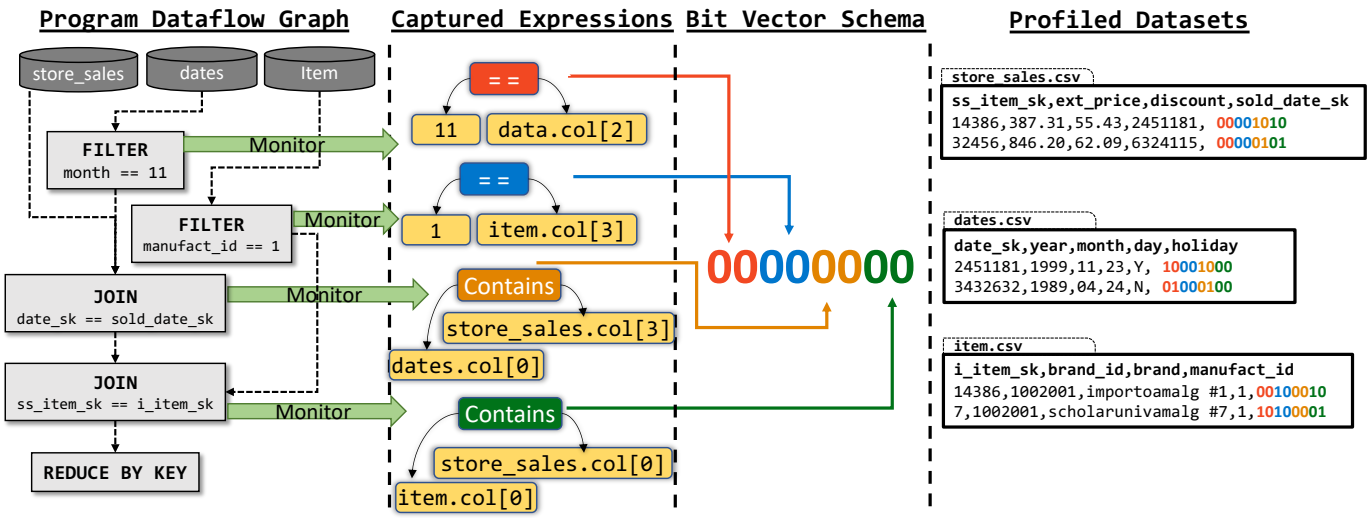



Fig. 4: Branch predicate expressions, bit vectors, and augmented datasets constructed by NATURALFUZZ.

```
case class TaintedInt(value: Int, t: Taints, expr:
    Expression){
// overloaded + operator
 def +(rhs: Int): TaintedInt =
  return new TaintedInt(value + lhs,
                        t,
                        expr.add("+", rhs))
 def +(rhs: TaintedInt): TaintedInt =
  return new TaintedInt(
                value + rhs.value,
                union(t,rhs.t)
                rhs.add("+", rhs.expr)
            ) // more overloaded operators
```

Fig. 5: Taint analysis enabled Int type in Scala

as `Tree(Node(a), Node("+"), Node(b))`. Figure 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 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. 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 [19], 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 shows the output of this first step.

### C. 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 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, 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. We use two bits to encode the results of a single

**Algorithm 1:** Seed Input Selection Algorithm

```
Input: A set of datasets D
Output: A set of minimized datasets D_min
D_aug ← computePathVectors(D)
D_min ← {}
V ← 0
for d ∈ D_aug do
    d_min ← {}
    for row ∈ d do
        v ← getPathVector(row)
        if V ≠ (V | v) then
            if d_j ∈ D_min such that d_j.join({row}) ≠ ∅ then
                d_min ← d_min ∪ {row }
                V ← V | v
            end
        end
    end
    D_min ← D_min ∪ d_min
end
return D_min
```

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.

### D. 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.

*a) 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.

*b) 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.

## IV. EVALUATIONS

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?

*a) Benchmarks:* We use a commercial benchmark suite called the TPC-DS benchmark [14], 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 [20]. We present the results from eight out of the 99 TPC-DS queries. Prior

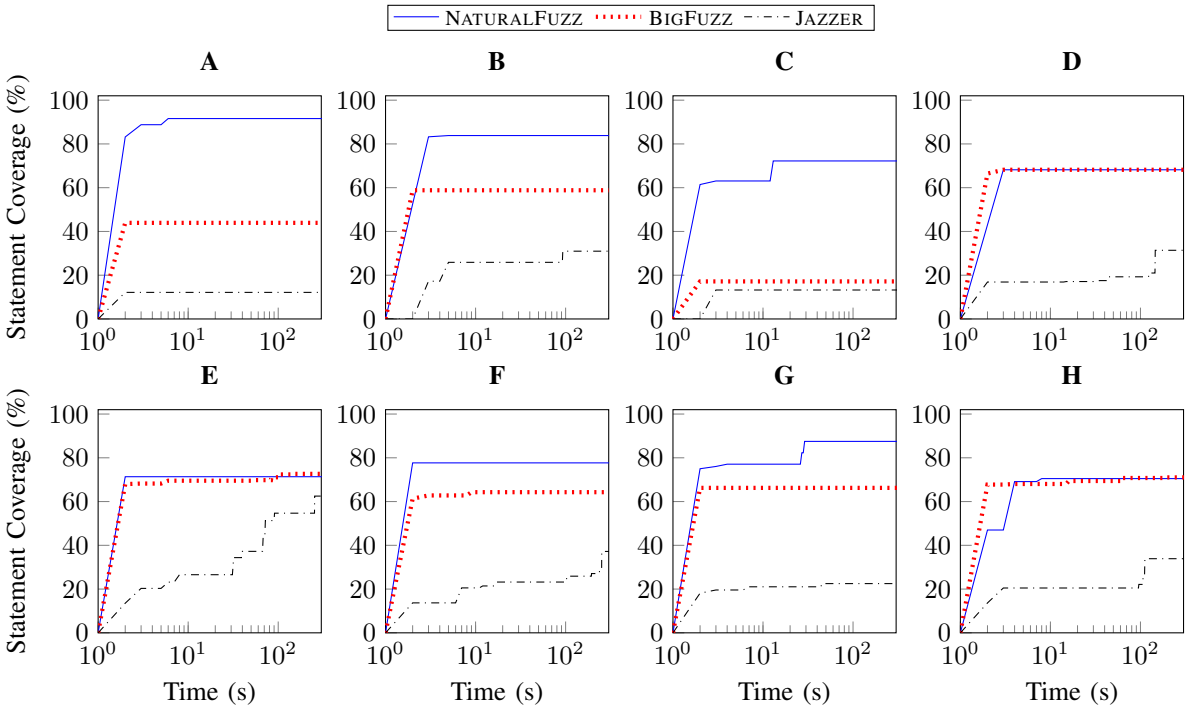



Fig. 6: Statement coverage progress of NATURALFUZZ and baselines on eight benchmark programs.

| 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 |

TABLE I: Benchmark programs and their descriptions as taken from the TPC-DS Specification [14]

work DISC application testing has mostly used simple custom applications, such as WordCount [4], 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 [14]. We translate the eight TPC-DS queries to Scala-based Apache Spark Applications, where every application ingests at least three datasets. Table I shows the programs used in our benchmark, including the number of datasets they use and their official description per the TPC-DS specification [14].

*b) 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 [13], 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.

| 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** |

TABLE II: Average overhead and average total errors detected by each tool.

*c) Fault Injection:* We perform mutation testing [21] 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 [4]. We further eliminate any experimental variations by averaging our results over five runs. In total, we run approximately 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.

### A. Code Coverage

Figure 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 I, 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.

```
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 */ }
```

Fig. 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.

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 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 [18] 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 to similar reasons.

### B. 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 II shows the average number of faults detected by NATURALFUZZ and

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

TABLE III: Comparison of coverage, fault detection, generation time, and generation size between SDV and NATURALFUZZ

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.

### C. 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 [13], a machine learning-based synthetic data generator tool. We configure SDV to use Conditional Tabular Generative Adversarial Network (CTGAN) [22] 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 III 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.

*a) 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. 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.

*b) 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

| | 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 |

TABLE IV: Average naturalness scores of inputs generated by NATURALFUZZ and baselines.

total on average. However, it is important to note that SDV is generating a single input containing 25K rows per dataset. Table I 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.

*c) 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.

### D. 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 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 [17], 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 [23] and DistilBERT, a lightweight version of Google's BERT model [24]. 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 IV 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.

### E. 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.

## V. RELATED WORK

**Data Generation for DB Testing:** The closest line of work to ours is database testing [12], [25], [26]. Houkjær *et al.* [26] propose a graph-based method to guide data generation. Bruno and Chaudhuri [25] propose a language-based technique for general-purpose data generation. Other DB test generation targets schema coverage [12]. These tools generate synthetic data with the goal of testing database performance

and workloads, and none of these target DISC applications. In contrast, NATURALFUZZ utilizes real-world data to generate minimal and realistic test inputs with the goal of discovering program faults.

We compared our work against ML-based fake data generator, SDV [13], similar albeit more primitive synthetic data generators exist such as Faker [27] that have been used to generate fake data in the literature before [28]. However, these synthetic data generators require manual configuration and do not take into account the complex constraints introduced by dataflow operators in the program.

**DISC Application Testing:** Prior work has also explored challenges in testing DISC applications. BIGFUZZ [4] is a black box fuzzing technique that uses schema-aware mutations to mitigate the problem of trivial parsing errors and increase coverage. It also uses framework abstraction to allow local fuzzing of the DISC application. BigTest [29] was the first to explore framework abstraction for testing DISC applications using symbolic execution. DepFuzz [30] aims to find co-dependent regions within the input and apply mutations that respect these co-dependent relationships. NATURALFUZZ's branch profiling technique is inspired by DepFuzz. All these techniques aim to maximize code coverage and fail to generate natural and intelligible inputs.

**Semantic and Grammar-based Testing:** Another related line of work is automated testing to find semantic faults. Zest [31] relies on program feedback to find semantic faults in the program and avoid 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 [32], [33], [34], [35], [36]. However, these techniques require the presence of grammar rules that are complex to write or infer. In contrast, NATURALFUZZ does not rely on any grammar rules.

**Taint-Based Fuzzing:** Several works have attempted to use taint analysis to identify regions of interest where mutations can be applied with priority. For example, Bekrar *et al.*[5] developed a fuzzing technique that uses taint analysis on program binaries to focus mutations on specific regions of the input. Similarly, PATA [8] performs path-aware taint analysis to mitigate problems of over-tainting and under-tainting. Other security-related works like BuzzFuzz [7] and TaintScope [6] attempt to isolate regions of the input that are used inside sensitive library calls to increase the chances of finding security bugs. These fuzzers do not target realistic input generation and employ tainting techniques that are not directly applicable to DISC applications since there is no single binary of a DISC application. Instead, the packaged binary is sent to thousands of machines where each runs separately.

**Profile-Based Fuzzing:** Profiling datasets and workloads is widely practiced across different streams of computer science [37], [38], [6], [39]. TaintScope [6] uses branch profiling to identify checksum fields and identify checksum integrity checks to bypass them. MoWF [39] uses branch profiling to determine if crucial branches have been explored. In contrast, NATURALFUZZ uses branch profiling to identify boolean expressions that guard code regions to facilitate realistic input generation.

## VI. CONCLUSION

In this paper, 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.

## ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation under grant numbers 2106420, 1764077, 1956322, 1460325, 2106383 and 2106404. It is also supported in part by funding from Amazon and Samsung. We want to thank the anonymous reviewers for their constructive feedback that helped improve the work.

## REFERENCES

[1] Apache hadoop. https://hadoop.apache.org/, 2022. Accessed: 2021-12-14.

[2] Apache spark. https://spark.apache.org/, 2022. Accessed: 2021-12-14.

[3] 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. IEEE, 2018.

[4] 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.

[5] 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.

[6] 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.

[7] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484. IEEE, 2009.

[8] 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.

[9] American fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2021. Accessed: 2021-12-14.

[10] 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.

[11] 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*, pages 475–485, 2018.

[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.

[13] 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.

[14] Tpc-ds version 2 and version 3, Accessed: 2022-09-01.

[15] Code Intelligence. Jazzer. https://github.com/CodeIntelligenceTesting/jazzer, 2022.

[16] Kostya Serebryany. Libfuzzer – a library for coverage-guided fuzz testing., Accessed: 2023-01-29.

[17] 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.

[18] 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*, pages 372–386, 2020.

[19] Spark programming guide, Accessed: 2023-05-04.

[20] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *VLDB*, volume 6, pages 1049–1058, 2006.

[21] 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.

[22] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. Modeling tabular data using conditional gan. *Advances in Neural Information Processing Systems*, 32, 2019.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Welcome to faker's documentation!, Accessed: 2023.

[28] 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*, page 806–817. Association for Computing Machinery, New York, NY, USA, 2021.

[29] 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.

[30] Ahmad Humayun, Miryung Kim, and Muhammad Ali Gulzar. Co-dependence aware fuzzing for dataflow-based big data analytics. In *Proceedings of the 31st ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[31] 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*, pages 329–340, 2019.

[32] 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.

[33] 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.

[34] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, jun 2008.

[35] P.M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, 1990.

[36] Emin Gün Sirer and Brian N. Bershad. Using production grammars in software testing. *SIGPLAN Not.*, 35(1):1–13, dec 2000.

[37] Mohammad Laghari and Didem Unat. Object placement for high bandwidth memory augmented with high capacity memory. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 129–136, 2017.

[38] Mohammad Laghari, Najeeb Ahmad, and Didem Unat. Phase-based data placement scheme for heterogeneous memory systems. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 189–196, 2018.

[39] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 543–553, New York, NY, USA, 2016. Association for Computing Machinery.