High-Level Mutations for JSON Typed Data in Big Data Fuzz Testing

L.E. Rhijnsburger, B.K. Özkan

Computer Science and Engineering
Delft University of Technology

June 27, 2021

Abstract

Fuzzing in Big Data applications is a relatively new field which is still lacking effective tools to support automated testing. Recently, a framework called BigFuzz was published which made fuzz testing for big data systems feasible. But there was no solution to work with Big Data programs that use JSON typed data. Big Data systems often make use of JSON typed data and JSON typed fuzzers for Big Data systems are currently not publicly found. With this work it is now possible to support JSON typed input data and apply fuzzing per iteration. The work requires a user defined input specification of the set of valid JSON inputs for the program under test, and a converted Java program based on the Spark program to test. However, it is almost certain the latter is not necessary in the future since it is likely this conversion can be automated.

This work is shown to be effective in finding bugs in a rather small amount of trials. Oppositely, it loses the descriptive exceptions, since it finds bugs later in the program instead of at the input validation phase. The work still has its limits to be applied extensively in the field of automatic testing, but serves as a proof of concept that automatically finding bugs in Big Data applications working with JSON typed data is in fact possible.

Keywords: json, input specification, input seed, json schema, unique errors, big data systems, bigfuzz, big data applications

1 Introduction

There is a huge growth in data in the last few years and it is touching almost all aspects or our life (Gupta & Rani, 2019). This rise in big data is hard to keep up with from a processable and computable kind of view, since testing Big Data systems poses quite some challenges (Steidl et al., 2020). Fuzz testing as a concept has already widely used but falls behind for Big Data. This is due to the fact that setting up automated tests for big data applications takes a lot of overhead, making the process very inefficient and time consuming. In fact, if fuzzing would be used like it is used today for Big Data applications, 98% of the time would go to setting up the test environment, and only 2% would go into fuzzing, which is highly inefficient (Zhang et al., 2020).

Recently, BigFuzz (Zhang et al., 2020) was published. BigFuzz is a tool that makes use of framework abstractions to be able to convert Spark code into an equivalent Java program. With this Java program, an extension of JQF (Padhye et al., 2019) is used to fuzz test the program with coverage guidance. The tool uses a user defined input, and can automatically test converted Spark programs.

However, the BigFuzz tool has its limitations. As of this moment it supports only tabular input, which keeps its application limited. BigFuzz was built as a tool to test data-intensive scalable computing systems (DISC systems), such as Apache Spark. Spark can use JavaScript Object Notation (JSON) as input and output structures in its pro-

grams. It is therefore only logical to be able to test these programs for possible bugs and errors. As of this moment there is no publicly known method to fuzz test Big Data applications that use JSON typed input data.

With this work providing support for JSON typed data, programs that make use of JSON inputs can also be tested with fuzzing and therefore contribute to more reliable and better Big Data programs. The main question of this research therefore is: "How can high-level mutations be implemented in a generic way for all kinds of JSON typed data with input specifications?". To answer this question, a closer look is taken to how this goal is achieved and how it is made effective. It is important to shed some light onto why this work is necessary in general. Moreover, the work must be effective to be used for automatic testing. Therefore it should be evaluated in reasonable settings. It is also interesting to look into a fully automatic variant of this work. Can this work be ran in the background without any user input? Finally, it is useful to look how and how well this work performs in comparison to other fuzzers that can work with JSON typed data.

For the evaluation of the implementation multiple benchmarks were used. These benchmarks are the same as Zhang et al. used in their research, but rewritten to make the programs support JSON typed input. With these benchmarks it is found that descriptions of bugs are somewhat lost, i.e. the bug report has less information about the specifics of the bug. Positively, the bugs found in this work are found in a rather small amount of iterations.

This paper is organized as follows. Section 2 provides a background of fuzzing with big data and input specifications for JSON. Section 3 describes the method used to answer the research questions. Section 4 elaborates on how the support of JSON typed input is added to the BigFuzz tool and how this work works. Section 5 explains how the benchmarks were used and how the effectiveness of this work was evaluated. Section 6 reflects on the reproducibility of the research. Section 7 summarises the results found within this research and its contribution to the research field, and Section 8 elaborates on the possible shortcomings and states some interesting parts of this research that could be worth giving more attention.

2 Background

For this research two topics are important to know about. Firstly, it is useful to know about input specifications for JSON typed data. Secondly it is necessary to take a closer look at the BigFuzz tool, since this framework will be extended with JSON support.

2.1 Input specification for JSON

Over the past few years, JSON has become the main data exchange format over the World Wide Web (Lv et al., 2018). For the scope of this research some definition is needed to define a set of JSON objects or arrays, and be able to validate if some object or array is valid within the restrictions given by the user or not. Some years ago a way of generalizing JSON objects and arrays was made: JSON Schemas. Wright and Andrews (2018) defined a JSON Schema document, or simply a schema, as a JSON document used to describe a JSON instance. A schema defines the types that can appear in a JSON instance and each defined type can have several properties. These properties vary from minimal or maximal lengths, to Regular Expression (Regex) patterns for strings, to complex object definitions with references to elsewhere defined values.

With the help of these schemas, it is now possible to validate a JSON instance against a schema. All properties of the instance are checked against the requirements from the schema which can distinguish an instance as valid, or invalid. While this is very useful, for the scope of this research the goal is not quite yet reached. It is necessary to be able to generate and mutate JSON instances. Fortunately, JSON schemas still have some use since the schemas can be used to generate a valid JSON instance as well. There are some minor implementations of this, but they lack either randomness of the generation, or completeness due to the fact that they only implement a single primitive element (e.g. booleans). The most complete implementation is json-generator (Dhua, 2019), which implementation correctly generates valid JSON instances, but lacks randomness. This library was therefore extended to support more random input generation (Rhijnsburger, 2021).

2.2 BigFuzz

BigFuzz was build as a tool to test Data-Intensive Scalable Computing systems (DISC systems), such as Apache Spark. The BigFuzz algorithm converts the given Spark program to a Java program, which does exactly the same to the input data as the original program. It converts Resilient Distributed Datasets (RDD's) to ArrayLists holding instances of the items that were in the RDD (e.g. a row of integers gets converted to an ArrayList holding integers). The converted program is then run with the input that is specified by the user. For example, a program can run some operation based on student numbers within a range. That range needs to be specified by the user before the program under test (PUT) is ran. This input specification is used by the tool to generate and later mutate the input parameters for the PUT.

BigFuzz is a grey-box fuzzing tool. It uses a form of guidance to find new branches in the code more effectively. It keeps track of when these new branches get covered, and saves those mutated inputs for later, to more in depth cover those new branches to find bugs and/or errors. The tool uses a run configuration which indicates what program to test, and how many loops it runs to find bugs and/or errors. BigFuzz prints its results to the console each iteration and saves iterations to a text file if that mutation resulted in an bug and/or error. The guidance that is used during a test run can be customised by the user as an extension of JQF (Padhye et al., 2019). The underlying tools that BigFuzz uses include JQF and JUnit (la Cruz Morales & Gwihs, 2000).

3 Approach

The aim of this research is to extend the framework to support JSON typed inputs. To support JSON typed data in the BigFuzz tool, a closer look to the code and the assumptions made by the framework, had to be taken. All implementations need to comply with the workflow and interfaces BigFuzz uses as well. Additionally a different data type is added to the framework, which means there needs to a be certain structure in the program to support the input, and eventually return a useful result. At last a comment is made about what full automation for this work would look like.

3.1 Contributing to the framework

There are certain interfaces and workflows within BigFuzz that enable fuzz testing. The important parts include the driver, the guidance and the mutation class.

The driver creates an instance of the guidance and starts up the program. It takes the arguments necessary for a run, which include the driver to use, the method to test and the amount of trials for the particular run. The driver needed almost no changed to work with JSON typed data.

The guidance is an important part of the BigFuzz framework, since it controls how and on what the mutations are applied and handles all situations where an exception is thrown. The guidance was hardly changed since this work focused on supporting a new data type and any changes to the guidance were not necessary to reach this goal.

The mutation class is the class where all the mutations to a test input are handled. It is used by the guidance to apply a mutation on a test input, but the class itself chooses what mutation is chosen each trial and processes these changes to the data.

All in all, to keep the framework as generic as possible, the driver and guidance are modified as little as possible, since these classes are also used to run benchmarks other than the JSON versions. The mutation class created in the guidance however, is made in a generic way to support all types of JSON input. To support the contribution of JSON within BigFuzz, the mutation class is made to implement the mutation interface as defined by BigFuzz.

3.2 Input assumptions

Supporting another input type to run the test framework with, is not trivial, because the whole BigFuzz framework is build to support their own variant of Comma Separated Value (CSV) files. Trivially the input files used in this work are JSON files. BigFuzz is able to work with multiple inputs and input files. This functionality should not be lost with this work and therefore some choices had to be made to make this possible. For example, an input file of the original benchmarks could contain multiple rows. This is due to the fact that the framework sometimes generates new rows for a given input, which are then saved as well after this trial. JSON objects can however

have many structures to support the same information. Some assumptions of the structure had to be made to make the sure this functionality of multiple rows was still supported. The work assumes that a single row, should be a JSON object, and these rows should be contained in a JSON array. This means that if an input file has 4 rows, the file would contain a JSON array with 4 JSON objects. The JSON object can be whatever the user wants it to be, but it should be specified by a JSON schema.

These JSON schemas are needed for two key points in this work. Firstly, during a run of the framework, the input file is always mutated, and there is a 50% chance that more input objects are generated during a mutation. For both these procedures, the JSON schema is necessary to consistently satisfy the assumed structure of the JSON input. When the input file is mutated, it chooses one of the possible mutations. One of the mutations from BigFuzz is the Data Distribution Mutation, which generates a new value for a given row and column, but ignoring the possibly specified range for this value. For example if the chosen row and column is a zip-code with the restriction of being in the range 900000 to 90099, a Data Distribution Mutation could mutate this value to 90100, which is not in the specified range. The schema is in this case (just like some other mutations) necessary to keep the type of the value that should be mutated, and use or ignore the allowed range of this value.

Secondly, for the generation it is trivial why the schema is needed. Some new valid instances of the input need to be created, and as described in Section 2.1, this can be done with the specified JSON schema.

3.3 Automation

The process of creating a schema could also be fully automated but this will be less effective due to the fact that an automated process can only create a schema which will be less strict or exactly as strict as the input schema defined by the user. This is due to the fact that it is very hard to optimally choose a schema for a set of inputs since there is a trade-of between succinctness and precision (Baazizi et al., 2019). Because a set of valid JSON inputs is needed and not one that describes a single instance of JSON optimally, this choice for the trade-of is very hard. Also, the input specification can not be omitted because of the guided mutations that work with the input specifications data.

This work is also unique in the fact that it supports JSON accompanied with guidance to find bugs. There are however other fuzzers out there that learn the specification of JSON input. For example (Mathis et al., 2020), which focuses on getting past the input validation stage. This is something this work already does automatically, since it has the user defined input specification.

4 JSON Support

The contribution of JSON support for Big Data fuzz testing sounds quite trivial, but the implementation had some obstacles to overcome to make sure no functionality is lost within this work which was present in BigFuzz. Moreover, this work should still be easy to use for users not knowing the details of what happens inside the work.

This paragraph will go over the two aspects of this work: 1) The input. What is expected from the user to be able to run the work, and 2) The mutations. What happens to the data given as input during a run, and how does it find bugs?

4.1 User defined input

As mentioned in Section 3.2, the user should specify the input specification in order to automatically test the program. Currently, the framework targets Java programs, since the BigFuzz transform module, which transforms Apache Spark programs to Java programs, is not working on the public repository of the original authors. However, it is likely to assume that this process can be automated in the future, as claimed by Zhang et al..

With a program to test, there are still two specifications needed from the user: the initial input seed and the JSON schema. The JSON schema describes the set of valid JSON instances that should be accepted by the PUT. This schema is necessary to be able to mutate the data ran through the PUT, and to generate more valid inputs in 50% of the iterations. The initial input seed, is an initial input ran through the program. This input is valid, meaning the program will be able to successfully parse it. The user can also choose not to define a initial input seed, or define an invalid

input seed. The effect of this is further elaborated in Section 5.

4.2 High-level mutations

Six high-level mutations as defined in the BigFuzz paper are converted to support JSON typed input as well. There are only some slight changes to the mutations since the data type of what must be mutated now has a different structure, namely JSON. The mutations that happen exactly like defined in the original paper (Zhang et al., 2020) are the following:

- The Data Distribution Mutation (M1)
- The Data Type Mutation (M2)
- The Data Column Mutation (M3)
- The Empty Data Mutation (M4)

The Data Format Mutation (M5) as defined in BigFuzz is completely removed. In the BigFuzz paper this mutation is defined by that it replaces a delimiter with a random other character. The files used in the original framework are semi-structured with comma separated values. JSON does not have a delimiter and the mutation can therefore not be applied. One could argue that JSON is also a semi-structured data type, but it does format its data differently. Although it is possible to change one of the special characters in JSON (e.g. { } or [ ]) to some other character, this would not be useful. When working with a program that takes in JSON as input data, it is trivial that it would not support something that is not a valid JSON value.

The Null Data Mutation (M6) is slightly different for another reason. In the original paper, Zhang et al. define this mutation as removing a column from the input and continuing with one less column. With JSON there are no columns, just objects with properties. The implementation of this mutation is therefore changed to remove a certain property from the input, to create the same effect as the Null Data Mutation on the tabular input from BigFuzz.

During the process of mutating, the user defined JSON schema is also used. It is parsed upon creation of the mutation class instance, and used in two separate places: 1) when a property needs to be selected at random to be mutated, and 2) when there need to be generated more rows. This is an addition to BigFuzz since it creates the possibility to create more effective mutations. The original framework does also apply the mutations in this way, but does not currently infer the type of a value, but has it defined by the programmer of the mutation class. The JSON mutation class is therefore more generic and able to support more kinds of input without having to alter the mutation code.

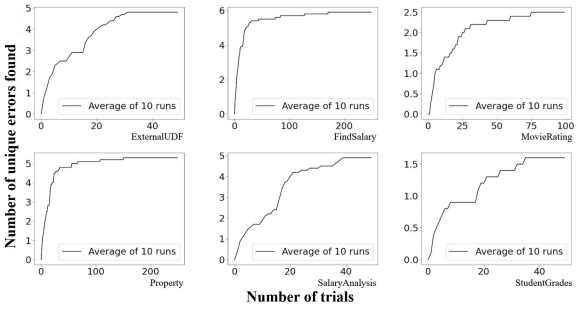

Figure 1: Average number of unique errors found against the number of trials on the benchmarks over 10 runs with a valid input seed

5 Evaluation

For this research some of the benchmarks from BigFuzz were rewritten to work with JSON instead of the tabular input from BigFuzz. They are considered as new benchmarks. The used benchmarks can be found in Table 3. Two empirical evaluations were run on the work of this research. First, The JSON contribution was run with the new benchmarks with a valid and invalid input seed. The results of this point out some effectiveness compared to the results of BigFuzz, but one important note should be taken into account. These benchmarks are completely different algorithms since they are rewritten to support JSON. One of the benchmarks might simply be better written, resulting in fewer bugs found by the framework or this work. Secondly, another way to evaluate the effectiveness and usefulness is to see what will be the difference when running it with a valid input seed or without an input seed at all. When the program does not receive any initial input seed, it will generate a valid one by itself, resulting in a different valid input seed for every run.

Exception	Cause
`ArrayIndexOutOfBounds`	Array access at incorrect index
`NumberFormat`	String parsing error
`StringIndexOutOfBounds`	String access with incorrect index

Table 1: Exceptions found on the benchmarks by BigFuzz

Exception	Cause
`ClassCast`	Double can not be cast to Long
	String can not be cast to Long
`NullPointer`	Lookup of a removed property
`NumberFormat`	String parsing error

Table 2: Exceptions found on the JSON benchmarks

P1	ExternalUDF
P2	FindSalary
P3	MovieRating
P4	Property
P5	SalaryAnalysis
P6	StudentGrades

Table 3: The benchmarks used for the evaluation

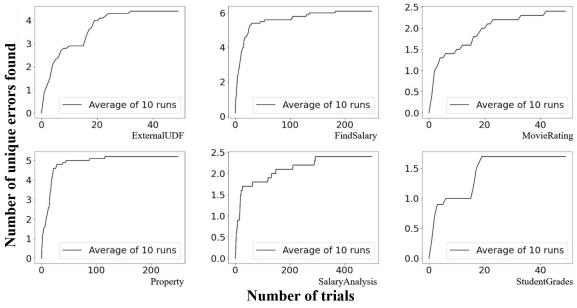

Figure 2: Average number of unique errors found against the number of trials on the benchmarks over 10 runs with an invalid input seed

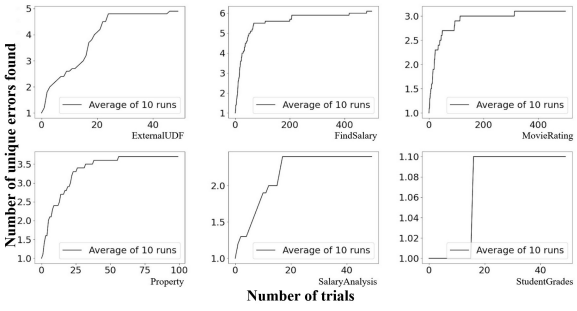

Figure 3: Average number of unique errors found against the number of trials on the benchmarks over 10 runs with a random input seed

5.1 This work compared to BigFuzz

The JSON contribution yields some interesting results compared to BigFuzz. It was expected that the JSON benchmarks should find the same bugs as the tabular version of BigFuzz, but this is not the case. The BigFuzz variant finds the Java Exceptions stated in Table 1, and the JSON work finds the Exceptions as seen in Table 2. This difference can be explained by the different way of parsing. Parsing errors are out of scope for this research, but below an explanation is given for the difference in the errors found by the framework.

BigFuzz handles all input as strings, and parses those strings in the program itself to the corresponding type. For example, a zip-code 90024 is stored and parsed into the program as a string. Only when this column is needed for calculation it is parsed to an integer. With JSON, the input is parsed as a JSON object, with each type immediately inferred from what is present. Meaning that if a zip-code property is an integer, it is stored and parsed as an integer. However when it is mutated to contain a character (e.g. 900#24), it is immediately stored as a string, and also parsed again as a string. The program however still assumes it will get an integer from this column, resulting in a `ClassCastException` when retrieved from a JSON object. This explains the difference in the `ClassCastException` and `NumberFormatExceptions`. An exception to this is P2, which still needs to parse an integer from a string, since it accepts integers prepended with the $ sign as well.

The `NullPointerException` is linked to the `ArrayIndexOutOfBoundsException`, since these both happen when a column or property is removed from the input rows or object. BigFuzz tries to access an index in an array to access the column where a value should be present, and the JSON contribution tries to access the value of some property which is no longer in the JSON object, resulting in a null value. The `StringIndexOutOfBoundsException` corresponds to the `ClassCastException`. This happens in BigFuzz when a column is replaced with an empty string, which will

result in the said exception when it tries to parse a value out of this empty string. In this work this results in a `ClassCastException` again since there is a string in a place where another type of value is expected.

5.2 Valid input seed vs invalid input seed

Since the user has the option to specify an input seed, it is useful to see how well this work performs when given a valid input seed versus an invalid input seed. To compare these two options, the framework is run 10 times, keeping track of the number of unique errors found in which amount of trials, for a valid and invalid initial input. An invalid input can be defined in two ways: 1) an input that makes the program crash, or 2) an input that does not make the program crash, but is not within a valid range as defined by the user. The way the framework is built, a user can define multiple input objects, which are both used during a test run. Because of this it was chosen to use both options in the invalid input test runs. The results this can be seen in Figure 1 and 2. As can be seen in these Figures, using a valid input seed mostly outperforms or matches the invalid input seed. For a single benchmark, P6, the invalid input seed performs only slightly better than the valid input seed. It is also visible that the valid input seed converges either quicker or as quick as the invalid input seed does. Since the valid input seed either performs better or as well as the invalid input seed, with a single exception to the rule, the use of a valid input seed is preferred over an invalid input seed.

5.3 Valid input seed vs random input seed

Another way to look at the work, is to look at how well it performs with a random input seed. This random input seed is created from the JSON schema which is specified by the user. This generated input seed will also be a valid one, but this time its a random input seed instead of one chosen by the user. The results of each benchmark ran with a random input seed can be seen in Figure 3, and can best be compared with the valid input seeds from Figure 1. When comparing the results, it is seen that the random input seed is mostly outperformed by the valid input seed. The only case the random input seed performs better is P3. On all other benchmarks the valid input seed either converges faster or finds more bugs on average. Looking at effectiveness, it is therefore concluded that a valid input seed works better than a random input seed. However, a positive aspect from the random input seed can be that you do not need to have a user defined valid input. Once the user has a schema, it can run the JSON fuzzer without any issue, only with less optimal results.

6 Responsible Research

During this research, the best effort was made to make sure this research was reproducible and that it could be validated by others. The evaluation of this work is mostly based on empirical results compared to itself. This is due to the fact that there were some issues in reproducing the results from BigFuzz. This section will go into detail about the reproducibility of BigFuzz and the reproducibility and validity of this work. After that a small comment is made about automated testing with this work.

6.1 Reproducing BigFuzz

There are some comments to be made about the reproducibility of BigFuzz. The repository linked by the BigFuzz paper (Zhang et al., 2020), was quite different from what is claimed in the paper itself. Several parts of the framework did either not run at all, or ran in a way that the paper did not describe it. An example of the latter is that the BigFuzz paper talks about schema-aware mutations, while the mutations in the repository happen with benchmark specific mutation classes which are not schema-aware. Since it was not in the scope of the project to remake the repository as described by Zhang et al., no effort went into fixing this. Bear in mind, that this is no criticism towards the authors of BigFuzz, since it could very well be that they did made a mistake in uploading the latest version of their work or do not want to publish their final work due to some unknown reason.

6.2 Reproducibility and Validity

During this research, a big effort was made to make this work reproducible, so that interested

readers or researchers are able to validate everything that is claimed by this paper. The most helpful step is that the full repository is published publicly (van den Berg et al., 2021). Furthermore, the work executes in such a way that after each run, the output is stored in a structured way. It contains all input used for each generation, in which trials what amount of unique failures were found and which unique failures were found specifically.

Moreover, the parts that the user should specify before a run, are either explained or referenced by the repository, such that a user knows what to specify or make before a correct run can be made. An important part about this is the Java converted Spark program. BigFuzz claimed they automated this part, such that a user only has to specify a Spark program and an initial input to run the framework. As mentioned in Section 6.1, not all parts worked in their published repository, of which also the transformation module, as mentioned in Section 3.2. Because of this, these equivalent Java programs have to be made by the user. This work has done so for 6 benchmarks from BigFuzz, and these are well documented, such that any user with programming experience is able to make the tool work with their own program.

6.3 Automated testing

Another important comment on this research is its usefulness. The framework and this work are meant to find unexpected bugs in programs. The keyword here is unexpected, since the framework really works the limits of what programs can and cannot handle. For example the framework finds bugs for missing values or incorrect formats of values. This is unexpected input and is what BigFuzz was intended to do according to Zhang et al.. The comment that should be made here, is that the framework is biased to find certain errors and lacks a bit of focus on the somewhat more trivial cases. For example when a program takes and integer as input somewhere without any size limits, the framework would almost never try possible edge cases or integers, usually being 0 or 1. These trivial edge cases exist for more types of input, and are found very rarely by BigFuzz or this work. Therefore it is very important to keep in mind that the framework, as well as this work, is not meant to replace all testing procedures. It should still be used along with user tests or some other tool.

7 Conclusions

Fuzz testing for Big Data that works with JSON was not yet automatically possible. With this work this process can be partially automated. Given an initial input, a JSON schema as input specification, and a Java conversion of the Spark program to test, the said work is able to find bugs in rather small amount of time compared to BigFuzz which is very well suited for quick semi-automated testing. Empirically, it is shown that the JSON contribution works best with a valid input seed. Nonetheless, a drawback is that the bugs found by this work, are less descriptive than the bugs found by BigFuzz. Moreover, it is not possible to automate the input schema generation, due to the lack of precision of a user defined schema. An automatically generated JSON schema will not cover the set of valid JSON input as well as a user is able to do so. Besides, this work also is very well able to get past the input validation stage because of the input specifications given by the user, which ensures that the testing done by the framework actually tests the program of the user.

8 Future work

The work from this paper is a promising contribution to Big Data testing in the future. The extension of testing JSON typed programs as well means the tool is closer to be used as a generic tool for all types of Big Data programs. However, several aspects can still be improved. The types of mutations in this research come from the BigFuzz tool and no effort went into looking for better or more effective mutations, while this is very interesting since JSON is a whole other data type than tabular data. Furthermore there is still a lot of progress to gain on the coverage guidance for the framework in general, and especially while working with specific data types. The guidance works fine with the small benchmarks used in this project, but interesting results may follow from in depth research into specific guidance for JSON typed data.

References

Baazizi, M.-A., Colazzo, D., Ghelli, G., & Sartiani, C. (2019). A Type System for Interactive JSON Schema Inference (Extended Abstract). In C. Baier, I. Chatzigiannakis, P. Flocchini, & S. Leonardi (Eds.), *46th international colloquium on automata, languages, and programming (icalp 2019)* (Vol. 132, pp. 101:1–101:13). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. Retrieved from `http://drops.dagstuhl.de/opus/volltexte/2019/10677` doi: 10.4230/LIPIcs.ICALP.2019.101

Dhua, J. (2019). json-generator. Retrieved 2021-05-26, from `https://github.com/jignesh-dhua/json-generator`

Gupta, D., & Rani, R. (2019). A study of big data evolution and research challenges. *Journal of Information Science*, *45*(3). doi: 10.1177/0165551518789880

la Cruz Morales, J. L. D., & Gwihs, S. (2000). JUnit. Retrieved 2021-05-27, from `https://junit.org`

Lv, T., Yan, P., & He, W. (2018). Survey on JSON Data Modelling. In *Journal of physics: Conf. series.* doi: 10.1088/1742-6596/1069/1/012101

Mathis, Björn, Gopinath, Rahul, Zeller, & Andreas. (2020). Learning input tokens for effective fuzzing. In *Issta 2020 - proceedings of the 29th acm sigsoft international symposium on software testing and analysis.* doi: 10.1145/3395363.3397348

Padhye, R., Lemieux, C., & Sen, K. (2019). JQF: Coverage-guided property-based testing in Java. In *Issta 2019 - proceedings of the 28th acm sigsoft international symposium on software testing and analysis.* doi: 10.1145/3293882.3339002

Rhijnsburger, L. (2021). json-generator-v1.0. Retrieved 2021-05-26, from `https://github.com/LarsRhijns/json-generator`

Steidl, M., Breu, R., & Hupfauf, B. (2020). Challenges in Testing Big Data Systems: An Exploratory Survey. In *Lecture notes in business information processing* (Vol. 371 LNBIP). doi: 10.1007/978-3-030-35510-4_2

van den Berg, B., van Koetsveld van Ankeren, L., Rhijnsburger, L., Smits, M., & Oudemans, M. (2021). bigfuzz-json-contribution. Retrieved 2021-06-07, from `https://github.com/LarsRhijns/Json-Fuzzer`

Wright, A., & Andrews, H. (2018). JSON Schema: A Media Type for Describing JSON Documents. Retrieved 2021-05-26, from `https://json-schema.org/draft-07/json-schema-core.html#rfc.section.1`

Zhang, Q., Wang, J., Gulzar, M. A., Padhye, R., & Kim, M. (2020). Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction.. doi: 10.1145/3324884.3416641