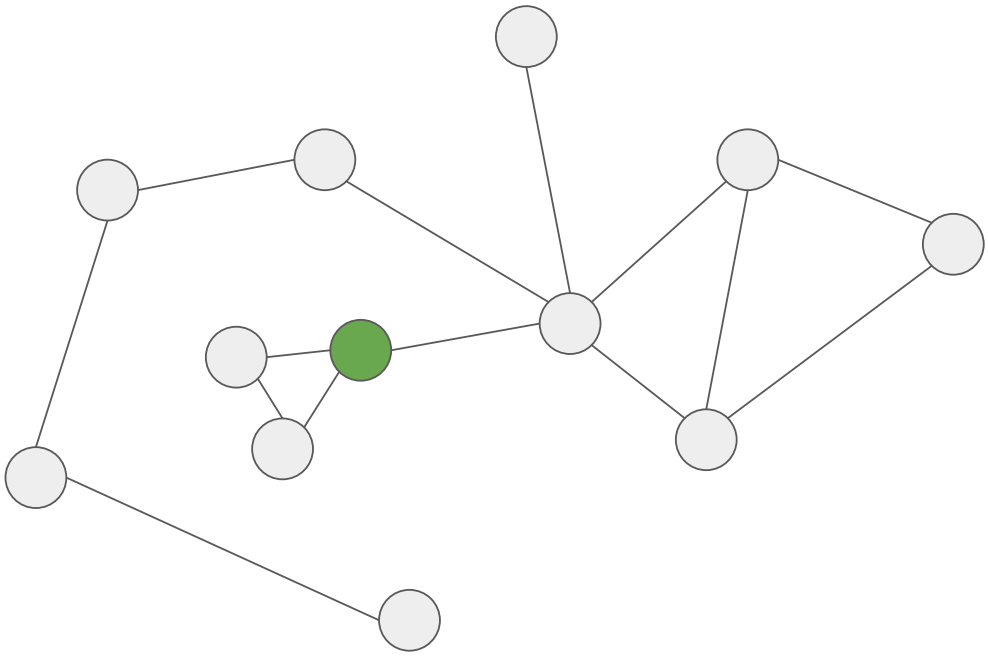
PGFuzz: Coverage Guided Testing of Graph Processing Applications

Master's Thesis



Melchior Willem Marcus Oudemans

PGFuzz: Coverage Guided Testing of Graph Processing Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Melchior Willem Marcus Oudemans
born in Amstelveen, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

PGFuzz: Coverage Guided Testing of Graph Processing Applications

Author: Melchior Willem Marcus Oudemans
Student id: xxx
Email: m.w.m.oudemans@tudelft.nl

Abstract

The rise of graph processing has led to an increase in the usage of graph databases and the availability of various frameworks. Graph databases have become more accessible and, in specific instances, can compete with relational databases. Testing an application with a relational database backend has shown limited test coverage, and current test generators cannot cover every branch condition in graph processing applications. There is a lack of test methods specifically designed for applications that utilize graph structures.

This paper presents PGFuzz, a coverage-guided, schema-aware fuzzer for graph processing applications. PGFuzz utilizes existing graph generators to generate inputs and applies graph-specific mutations to alter the graph state. These mutations are schema-aware, designed to cover the graph model search space and satisfy logical conditions from real-world applications. The mutations involve adding new graph elements, removing graph elements, modifying existing elements, altering property values, and violating graph constraints. When compared against existing graph generators and a random byte mutation approach on the nine real-world examples in our benchmark suite, PGFuzz demonstrates an increase in coverage over time and detects more logic errors than the other methods. PGFuzz can cover all previously uncovered branching.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	B.K. Özkan , Faculty EEMCS, TU Delft
External supervisor:	S.G. Dumbrava, ENSIIE & Télécom SudParis
Committee Member:	Dr. J.G.H. Cockx, Faculty EEMCS, TU Delft

Preface

I would like to express my gratitude to the following individuals for their invaluable support and guidance during the completion of this project. Firstly, I am indebted to Burcu Özkan and Stefania Dumbrava for their daily support and guidance. Burcu Özkan's expertise and advice were instrumental in structuring my project, and Stefania Dumbrava provided the motivation and encouragement that kept me going during challenging times.

I am also deeply thankful to my parents for their support and belief in my educational journey. Additionally, I extend my heartfelt appreciation to Sophie Vredenbregt for standing by my side throughout this challenging year. Lastly, I would like to express my gratitude to Dylan Rijlaarsdam and my circle of close friends, whose personal support has been invaluable. Everyone's encouragement and assistance have made a significant difference.

Melchior Willem Marcus Oudemans
Delft, the Netherlands
July 8, 2024

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Every year, the amount of data stored is growing, reaching already the Peta- and Zetta bytes [18]. To process these quantities, frameworks and methods must be tailored to the data characteristics. One of the growing data types is interconnected data, requiring graph processing methods [47]. The current approaches to processing graph data have shown to solve previously impossible tasks and provide better performances [37, 48, 61]. Some widely known adaptations of graph processing are in fraud detection with the analysis of the Panama papers [52, 53], analyzing tools during the COVID pandemic[40] and Google's page rank search engine [7]. Graph processing is also actively applied in other areas like social media networks, biomedical domain [58], AI [57], and more[41].

With the rise of graph processing, the usage of graph databases has increased, and with it, the number of available frameworks [51]. These engines are specifically designed for interconnected data and facilitate a more efficient traversal method between this kind of data compared to traditional relational databases [56]. Graph databases have become more accessible and, in specific instances, can compete with relational databases [20]. Applications using a relational database have shown to have limited test coverage for methods that interact with the database [62], and similar issues might arise with graph databases.

The development of graph processing applications can benefit from more mature testing tools. Software testing is generally considered an important step in development, and automated testing is widely used. The research done in graph processing has primarily focused on solutions to tasks mentioned earlier and performance evaluations [5, 15, 20, 55, 60]. The work on testing graph databases targets the framework engines rather than the applications using these frameworks [27, 30, 64]. There currently seem to be no methods designed specifically for testing applications that use graph structures.

Fuzz testing has already been shown to be an effective automated testing tool that can be applied to many different areas [34, 66]. Fuzz testing relies on generating lots of input

and running it on an application to traverse as much of the application as possible. As far as we know, there is not yet a fuzz method for applications using graph structures. This work presents PGFuzz, a coverage-guided, schema-aware fuzzer for graph processing applications. PGFuzz increases the application exploration using input from existing graph generators and graph schema-aware mutations. The mutations are designed to cover the graph model search space and satisfy logical conditions from real-world applications. We make the following contributions:

- PGFuzz, a coverage-guided, schema-aware fuzzer for applications using a graph database backend.
- A benchmark suite collected from real-world applications containing graph structures and graph processing.
- Comparison with a naive mutation strategy using a random byte mutation approach for graph structures that reduces the number of corrupted database states.

We have evaluated PGFuzz on a benchmark suite created from applications found in public repositories and other literature. PGFuzz is compared against existing graph generators to see the exploration increase (i.e., coverage increase) and the unique error detection over time. The effectiveness of the PGFuzz mutations is compared against a random mutation approach. Also, the effectiveness of each mutation strategy is evaluated by tracking individual contributions.

The paper is structured as follows: In section 1.2, more information surrounding graph processing, database testing, and fuzz testing is given. Section 3 presents PGFuzz, containing the framework, graph model, mutation design, and fuzz guidance. In section 4, the performance of PGFuzz is presented. Section 5 discusses related works like graph generation, database testing, and fuzz testing. Lastly, in section 6, the conclusion, discussion, and future works are discussed.

1.2 Background

There are many different approaches to processing the interconnected data. Almost every programming language has some graph processing library, be it in an object-orientated programming language (e.g., GraphQL for Java [1]), a functional programming language (multitudes for Python[2]) or a distributed system (e.g., GraphX for Spark [3]). Graph databases are specifically designed for large interconnected data sets. These databases have similar design principles as traditional relational DBMSs but store the data differently. As these databases can provide data guarantees similar to those of traditional systems, they can also be used for applications to manage their data. There are already many different implementations of

[1] https://www.graphql-java.com
[2] https://wiki.python.org/moin/PythonGraphLibraries
[3] https://spark.apache.org/graphx

database models, some of the most popular in 2024 being Neo4J, Memgraph, JanusGraph, and TigerGraph [51].

Multiple works have already explored testing the graph databases, making queries, and applying differential testing to find bugs [22, 27, 64]. The most important parts of Graph DB engine testing are making proper graphs following a schema and creating (complex) queries. Other works further explore the query generation approach by, for example, query partitioning [30]. Most of these works randomly generate their graph schemas, selecting an arbitrary amount of vertices, labels, edges, and properties [22, 64]. Another approach is to use a meta-model of the graph, allowing the generation of specific labels with a predetermined distribution [30]. The last example uses an abstract graph summary of the current graph to determine the schema and make further modifications to the state [27]. They do not restrict the generation to additional constraints. Applications using these database engines might require more complex semantics, following specific structures or implicit dependencies.

Methods to test applications using databases have specialized their approach to account for these specific structures and constraints, minimizing the amount of invalid database states [13, 16, 17, 62, 65]. These methods are specifically designed for relational databases as their approaches vary, from dynamic user input, to schemas, or specialized table mutations. These methods are specific to relational databases, making them not directly applicable to graph databases.

1.3 Motivation

As with any kind of automated testing by exploring the application, testing graph processing applications requires a system that generates inputs. For an application that takes a string as input, generating inputs is easily achieved by generating random characters. The generation becomes less trivial when certain semantics are expected (e.g., email addresses like @domain.com). Graphs have a complex structure, allowing for different nodes, edges, and properties. Graphs become even more complex when constraints like uniqueness, non-null, or cardinality are defined. Existing graph generators allow some of these complexities to be modeled in a schema, only generating graph states that fit the requirements.

Applying graph generators in automatic testing presents two challenges. First, graph generators only generate valid graphs, which might not cover every branch of an application. The second challenge is that most graph generators produce independent database states, not utilizing the coverage information gained from a previously generated state. This might increase the initial coverage results as the search space is more generally sampled, but more complex and nested branching would be less likely to be found. The challenges will be further discussed in the following three examples. Each example highlights a different graph component that can determine the branching in an application, the first being cardinality constraints, the second edge labels, and the last properties and their values.

The code fragment shown in Listing 1.1 contains branching logic dependent on a cardinality constraint. In the original application from which it is collected, it can be used to ensure a patient has exactly one way of contact for a visit. The procedure *manageRelationships* is called when a value node needs to be updated to a new item, removing any connections with the previous item. The function *manageRelationships* collects any existing relationships which should be maintained and verifies whether the original state is valid. Different paths are taken if the previous or new state with the new item is invalid. This example is a one-to-one constraint, making a relationship unique for two node types. A graph generator would only generate states where the connected nodes have a single relationship defined. The branching defined within the if conditions would never be reached, lowering the overall coverage. If the graph generator did not use the cardinality constraints, passing the first cardinality check would become less likely, again limiting the overall coverage.

The code fragment in Listing 1.2 is used to identify the travel methods from a location in a demand-responsive service implementation. The function *getTransport* takes a graph containing all locations, transportation methods, and travel times. The edges from a *centroid* (i.e., location) are looped over to see which transportation methods are available, assuming there is always a label with *"DRT"* or *"WALK"*. shows an implementation where the node and edge labels determine the branching. The conditions only consider two predefined values on the edge labels. If there is a node without any edges, the function will fail.

The code fragment in Listing 1.3 is collected from a system that analyses a large number of genomes. The function *preparePhasedGenomeInformation* prepares genome information, touching connected components and collecting the property values. In the fragment, the property values of the selected node are accessed and used to determine the branching. In this example, the graph must have specific properties to load the data correctly and access different branching. These properties need to have the correct key, type, and format. Additionally, the value of the property determines which branch path is taken. The chance of getting in the else branch of the if condition would be determined by the bounds of the property *phasing_chromosome*. Re-using and modifying an input that has reached that branch could increase the search performance compared to generating new independent inputs from a graph generator.

```
manageRelationships(Node prevItem, Node newItem, String value) {
    ...
    if (!isSingle(PrevItem, relationshipToMaintain) {
        ...
    }

    if (!isSingle(NewItem, relationshipToMaintain) {
        ...
    }
    ...
```

```
    if (!isSingle(valueRelationship) {
        ...
    }
    ...

}
```

Listing 1.1: Code example with cardinality constraints

```
getTransport(Graph g) {
    int transport_count = 0;
    ArrayList<String> trsp = new ArrayList<>();

    Node n =  g.getNodes("Centroid").get(0);
    for (Edge e : n.getEdges()) {
        if (e.getLabel().equals("DRT") || e.getLabel().equals("WALK")  )
            {
            trsp.add(e.getLabel());
            transport_count += 1;
        }
    }

    String transport;
    if (transport_count == 2) {
        transport= "DRT/WALK";
    } else {
        transport= trsp.get(0);
    }
    return transport;
}
```

Listing 1.2: Code example with branching logic depending on edges

```
preparePhasedGnomeInformation(Graph g) {
    ...

    chromosomeNr = (int) node.getProperty("phasing■chromosone")
    phasingID = node.getProperty("phasing■ID")
    if (chromosomeNr! = 0) {
        ...
    }
    else {
    ...
    }
}
```

Listing 1.3: Code example with branching logic depending on property values

Chapter 2

Fuzz Testing

This chapter introduces fuzz testing and discusses the considerations when applying fuzz testing to graph processing applications.

2.1 Fuzz Testing Methods

Fuzz testing is a common method to test applications automatically. It does so by repeatedly running the application on different inputs, as the inputs usually determine the application's behavior. Running the application on many different inputs should result in different execution paths. The challenge in fuzz testing is to generate as many useful inputs as possible. New inputs are usually created by modifying existing inputs, called mutations. Fuzzers are effective in discovering many of the internal states of an application and finding vulnerabilities like application halting exceptions [34].

Generally, fuzz testing is easy to deploy, has good extensibility and applicability, and can be applied with limited knowledge about the application being tested [34]. The fuzzer can run with no information about the application (black box), some information about the application (grey box), or with full knowledge of the inner workings of the application (white box) [66]. A white box method might know which conditionals have been covered for each test run, while a black box can only use the output value from the application.

One of the earliest and most widely used fuzzing methods is AFL [44]. It is a black-box fuzzing approach with often random bit mutations. While the method can be used for graph-like structures, the mutations are inefficient for complex structures, as further discussed in section 3.4. For a grey-box fuzzer, the fuzzer needs to be able to mutate inputs to utilize the feedback. While many different gray-box fuzzers exist, their mutations can not be directly applied to graphs as they are created for different kinds of data structures [23, 62, 63].

The following sections propose two novel approaches for fuzz testing applications requiring graph structures: a black box approach using graph generators and a grey box approach using graph-specific mutations.

2.2 Fuzzing graph structures

In black box fuzzers, there is no information about the internal states of a program and its functionalities [66]. The application is repeatedly run with different inputs, searching for different outputs and potential crashes.

We implemented a black-box fuzzing approach using graph generators. A graph generator is used to provide seed inputs for the application. Whenever the inputs are exhausted, new inputs are generated. This approach does not require the graphs to be mutated, as new inputs are generated from scratch. As a result, there is no feedback from the individual runs. Every input generated by the generator is independent of the others. This method can use any graph generator, allowing for the most relevant generator to be used for an application.

Figure 2.1 depicts a basic fuzzing framework with a feedback loop, commonly seen in grey-box fuzzers. The fuzzer starts with initial inputs provided in the seed. After applying input on the application, some information can be collected about the run, such as which branches have been covered. Using a fitness metric that determines the quality of the input, the input is either selected for future iterations or discarded. Inputs are then changed so that the input might satisfy different branching conditions in a future iteration. The mutations can either consist of small changes like a bit flip or large changes like changing an integer to its maximum value [66].

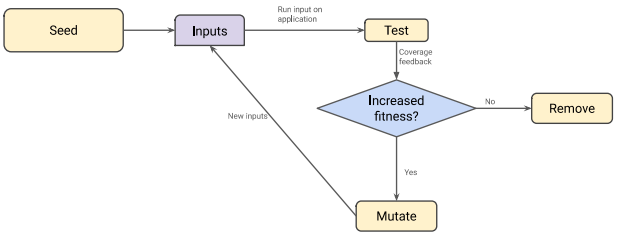


Figure 2.1: Basic grey-box fuzz loop

Fuzzers can have different kinds of mutations, each with a trade-off. More specialized mutations might explore the search space better while they are less widely applicable. Generic mutations might be generally applicable, but the application has not been explored well.

PGFuzz is the first grey-box fuzzer that describes mutations for graph-like structures. The mutations in PGFuzz are schema-aware, meaning the graph's structure, typing, and constraints are considered when a mutation is applied. We also implemented a random mutation approach based on random byte mutations, further discussed in section 3.4. While the random approach requires less input from the user, it can make fewer kinds of changes to a graph state.

Chapter 3

PGFuzz

This chapter introduces the different components of PGFuzz. First, the workflow containing the framework's core processes is shown. Then, the graph model is defined, after which the mutation design is discussed.

3.1 PGFuzz Workflow

Figure 3.1 shows the workflow of PGFuzz. The flow has three main processes:

- **Application Instrumentation** - An application is instrumented to provide coverage feedback. The instrumentation injects markers in the code, which report back to the fuzz framework when a code path is reached [42]. New coverage is detected by tracking which markers have been reached and comparing the reached markers per input cycle. The application can be transformed before the fuzz loop starts and is re-used every test iteration.
- **Input Generation** - The input is generated by a graph generator. The generator passes an initial seed to the fuzz loop, which can be used for new inputs during the fuzz loop.
- **Fuzz Loop** - The fuzz loop is a repeating iteration of selecting an input, running it on the instrumented application, and mutating the used input if new branching is discovered.

Figure 3.1 highlights the main fuzzing loop components using an orange square. PGFuzz, highlighted in the blue square, processes the coverage feedback and applies the mutations defined in Section 3.3. PGFuzz uses the coverage feedback received from the fuzz framework to guide the next iteration. An initial seed or inputs from a graph generator are needed, as PGFuzz does not generate graph states from scratch. If an input does not provide new coverage, it is discarded. Whenever there is new coverage, PGFuzz stores the used input and queues it to be mutated.

Queued inputs are repeatedly mutated until the predefined mutation depth is reached. The maximum mutation depth prevents the fuzz loop from unlimited mutations on the same state and allows for new exploration from the graph generator. The inputs from the queue are randomly selected, meaning PGFuzz has a breadth-first over the available inputs rather than a depth-first search.

PGFuzz is built on top of a fuzzing framework, JQF. This framework instruments the code and passes inputs from PGFuzz. PGFuzz uses an internal graph structure, meaning it can run with any kind of property graph structure. There only needs to be a translation layer between the application's inputs undergoing testing and PGFuzz.

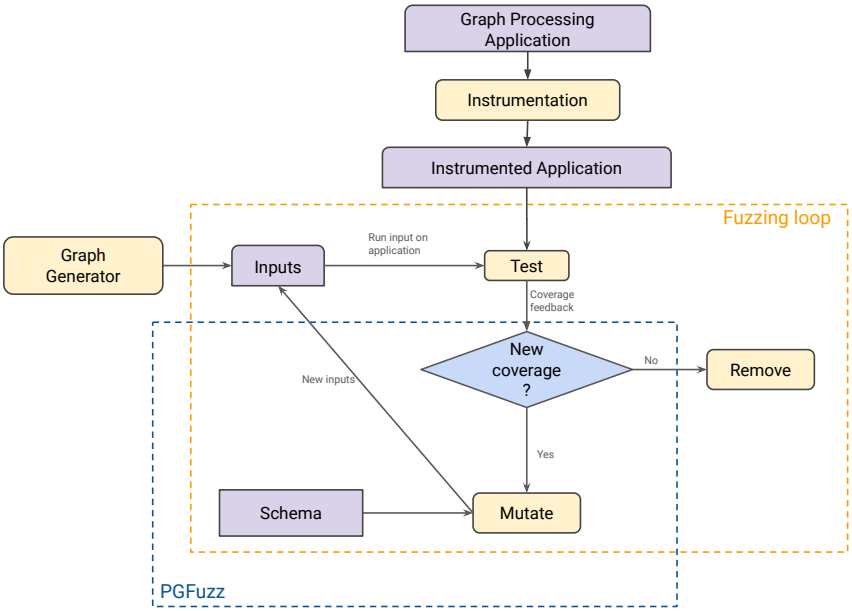


Figure 3.1: PGFuzz workflow

3.2 Graph structure

PGFuzz has an internal graph structure based on a property graph model, formally described by Bonifati [10]. It is a multi-edged, directed, and labeled graph with property values on both the nodes and the edges. A graph can have multiple nodes, edges, and properties. Multiple labels can be defined for each graph element. Nodes and edges can have multiple properties. Figure 3.2 shows an example of a graph representing two authors, a book, and the relationship between the elements.

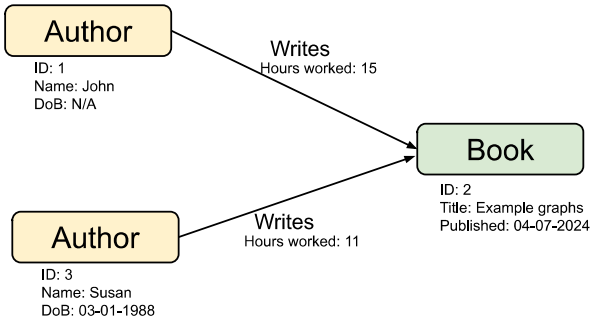


Figure 3.2: Example property graph

Besides these elements, PGFuzz also expects a graph schema. The graph schema describes the unique node labels, edges, and relationships. Additionally, the graph schema supports the following constraints:

- **Property Type** - Each property can have a type assigned. There are five built-in types (String, Integer, Double, Float, Boolean), which defaults to String when no type is provided.
- **Property Exclusiveness** - Each property can be flagged as exclusive. The property's value must be unique across all similar nodes and edges.
- **Property Mandatory** - Each property can be flagged as mandatory. When flagged, the property needs to have a value.
- **Relationship Cardinality** - The cardinality of a relationship indicates how many edges of one type are allowed between any two nodes. A cardinality can describe an in- and outnumber indicated by NxM or is described by *Simple*, *Multi*, *OneToMany*, *ManyToOne*, *OneToOne* [1].

Figure 3.3 visualizes a graph schema used by PGFuzz. The nodes represent the node labels, and the edge between two of the nodes represents the relationship between those nodes. A property is exclusive when it is visualized by bold text, and the italic text properties are not mandatory. The cardinality constraints are defined on the sides of the edge.

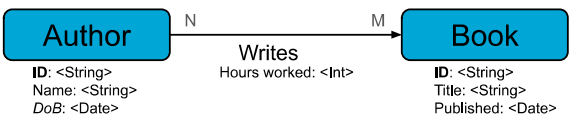


Figure 3.3: Example graph schema

[1] https://docs.janusgraph.org/v0.3/basics/schema/

Figure 3.4 shows a graph schema with node labels for employees and departments, and a worksAt relationship between those nodes. The ID property on both the employee and department nodes must be exclusive, and the 'DoB' property is not mandatory. The relationship 'worksAt' has a cardinality constraint of *manytoOne*, meaning a department can have many employees, but the employee can have only one department it works at. In figure 3.5, a valid graph state is shown which satisfies each constraint. Figure 3.6 is an example where the elusiveness constraint is broken, as the ID property in two of the 'Employee' nodes is identical. The mandatory constraint is broken in figure 3.7, as one of the 'Employee' nodes does not have a name property defined. Lastly, in figure 3.8, the cardinality constraint on the relationship is broken, as the employee John works at two different departments simultaneously.

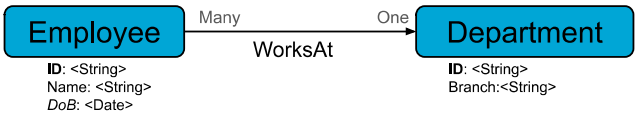


Figure 3.4: Graph schema with constraints

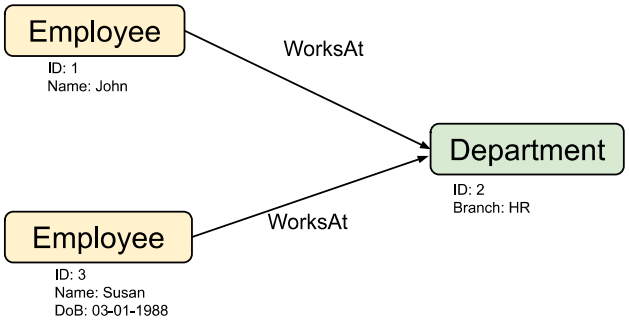


Figure 3.5: valid graph example

The graph schema PGFuzz implements is inspired by the PG-Schema, which was proposed in earlier works to provide a concrete schema model for property graphs [3]. The graph schema has a similar definition as PG-Schema, defining the graph elements with their contents and constraints based on PG-Keys, further discussed in section 3.3.

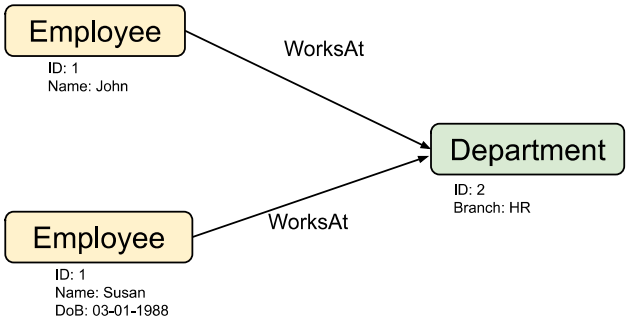


Figure 3.6: Broken exclusiveness constraint example

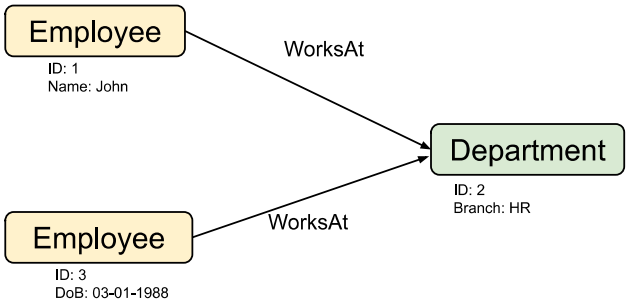


Figure 3.7: Broken mandatory constraint example

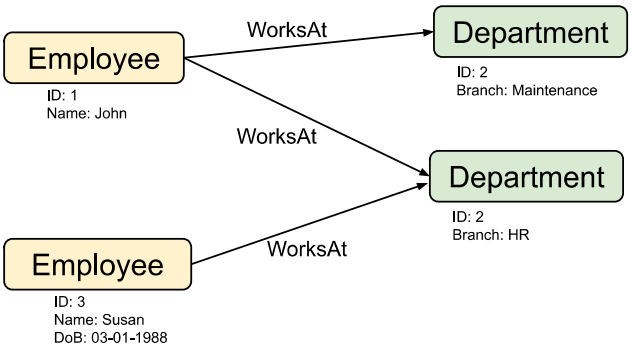


Figure 3.8: Broken cardinality constraint example

3.3 Schema-aware Mutations

PGFuzz uses graph-specific mutations instead of random bit— or byte mutations. These mutations preserve the graph structure and target specific graph elements to be changed. The mutations are schema-aware, meaning PGFuzz knows important characteristics of the inputs required. By using a graph schema, the mutator knows which graph elements can occur, what kind of relationships between entities can be made, and any constraints enforced on the input graph. Schema-aware fuzzers have been applied in other areas [59, 63], but

PGFuzz is the first to define it for graph structures.

Real-world applications have been tested with inputs from existing graph generators to see which branching could not be reached. The mutations have been designed to change the graph generator inputs to reach these branches.

Besides using these empirical examples, the mutations have been extended with theoretical constraints discussed in the work on PG-Keys. PGFuzz supports a core fragment of PG-Keys, which has informed the design of the novel GQL graph query standard [2]. The constraints used are cardinality and key type constraints. The cardinality constraint describes the cardinality of a relationship between any two nodes (e.g., One-to-One, Many-to-One). The key type constraint describes whether a property should be present (Mandatory) or unique (Exclusive).

The mutations used by PGFuzz are shown in table 3.1. The mutations are categorized by their effect on the graph as follows:

- **Add Graph Element (C1)** - Add a node, edge, or property to the graph.
- **Remove Graph Element (C2)** - Remove a node, edge, or property from the graph.
- **Change Graph Element (C3)** - Change a node, edge, or property in the graph.
- **Change Data Values (C4)** - Change a property value in a node or edge.
- **Break Graph Constraint (C5)** - Break one of the graph constraints.

More realistic changes can be made to the graph states using a schema. Rather than adding a random label, the mutator can select a specific label already expected to occur in the graph. Additionally, the graph schema allows for the mutator to know about constraints that might not be derived from any arbitrary graph state. Not every mutation requires the graph schema to make the change, as the information needed can be retrieved from the graph. The mutations using the schema try to follow the schema where possible but might deviate when there are no valid candidates to choose from. The mutations defined in C5 actively target breaking the schema constraints. Below, further explanation of the implementation details of each mutation are given:

- **Add Node (M1)** - Select a random node label from the graph schema. Create a new node with said label and add the required properties and edges to existing nodes.
- **Add Edge (M2)** - Select a random relationship from the graph schema. Select two nodes from the candidates' sources and targets and add a new edge with the required properties.
- **Add Property (M3)** - Select a random property from the graph schema and add it to a node or edge. Otherwise, generate a random property.

Category	Mutation	Name	Schema
C1	M1	Add Node	Y
	M2	Add Edge	Y
	M3	Add Property	Y
C2	M4	Remove Node	N
	M5	Remove Edge	N
	M6	Remove Property	N
C3	M7	Change Node Label	N
	M8	Change Edge Label	N
	M9	Change Property Key	N
C4	M10	Change Property Value	Y
C5	M11	Break Cardinality	Y
	M12	Break Exclusiveness Constraint	Y
	M13	Break Mandatory Constraint	Y
	M14	Change Property Type	Y
	M15	Remove Node Type	Y
	M16	Remove Edge Type	Y

Table 3.1: PGFuzz mutations

- **Remove Node (M4)** - Select a random node from the graph and remove it with any of its edges.
- **Remove Edge (M5)** - Select and remove a random edge from the graph.
- **Remove Property (M6)** - Select and remove a random property from the graph.
- **Change Node Label (M7)** - Select and change a random node label.
- **Change Edge Label (M8)** - Select and change a random edge label.
- **Change Property Key (M9)** - Select and change a random property key.
- **Change Property Value (M10)** - Select a random property from the graph schema and generate a new property value. Apply the generated value to one of the corresponding properties in the graph.
- **Break Cardinality (M11)** - Select a relationship with a cardinality constraint from the graph schema. Add new edges to one of the nodes in the graph to break the specified cardinality.
- **Break Exclusiveness (M12)** - Select one of the properties with an exclusive constraint. Either copy the element or select two random graph elements with the selected property and copy the value from one element to another.
- **Break Mandatory (M13)** - Select one of the properties with a mandatory constraint. Select a random property from the graph and set the value to null.

- **Change Property Type (M14)** - Select a random node from the graph and change the property value and type.
- **Remove Node Type (M15)** - Select a random node label from the graph schema and remove all nodes with said label from the graph.
- **Remove Edge Type (M16)** - Select a random node label from the graph schema and remove all nodes with said label from the graph.

3.4 Byte Mutations

Working from one or more initial states and applying changes to reach new branching in the code is the basis for fuzz testing. Fuzz testing has been widely applied on many different kinds of input structures and applications [34, 66]. One of the earliest approaches to fuzz testing was AFL, which applies random bit mutations. Random mutations remain an effective way to change the inputs and can, with the right guidance, keep up with more specialized mutation methods [65]. The method is less effective on larger and semantically complex data structures, as it either corrupts the data structure or does not have enough exploration to increase code coverage. This is well shown when mutating relational database system queries, which results in 70% being semantically invalid [65] and 90% when applied to big data analytics [63].

Applying random byte mutations to graph structures shows a similar issue. Storing an empty graph can already take up hundreds of bytes. Single-byte mutations cannot significantly change graphs as these require multiple bytes to be changed in different places in the structure. Adding a new byte, for example, takes around 57 new bytes and changing three existing bytes. For an edge, it is around 53 bytes, which also produces a shift in the bytes, changing every following byte after the location of the initial change. With a relatively small graph consisting of less than 100 nodes, edges, and properties, 87% of the graph structures are corrupted after a random byte mutation. Of the 13% that could still be interpreted, the changes made to the graph are primarily on the labels, property keys, and property values. This shows that the large mutations with multiple bytes are less likely to succeed than the smaller mutations on properties and label names.

Since the random byte mutations can only change the node labels, edge labels, and properties, we implemented another mutation method that changes these elements. This approach has the same mutation power as the random byte mutations but significantly reduces the number of corrupted states.

Chapter 4

Evaluation

This section will first discuss how PGFuzz is evaluated and how the results are collected. Then, the benchmark suite will be explained in detail. The results will be presented and used to answer the evaluation questions. Lastly, a case study about changing the mutation power via compound mutations is presented. The implementation of PGFuzz, benchmarks, and results can be found in the project repository[1].

To evaluate the performance of PGFuzz, we will answer the following questions

1. How effective is PGFuzz in increasing branch coverage compared to existing graph generators?
2. How effective is PGFuzz in finding unique errors?
3. How much does each mutation mechanism contribute to the increased coverage?
4. Which mutations are most effective in finding unique errors?
5. How does the performance of PGFuzz scale with different application complexities?

To answer RQ1, we compare PGFuzz's coverage over time with existing graph generators. For RQ2, we count the unique errors found by PGFuzz in a fixed amount of time. RQ3 is answered by running each mutation category independently and determining the individual contribution to the overall coverage increase. RQ4 is similarly answered by running each mutation category independently and counting the unique error occurrences. Finally, for RQ5, we doubled the branching complexity of one of the benchmarks and tracked the coverage over time.

4.1 Experimental Setup

We collected nine benchmark programs for the evaluation. Both the branch coverage and unique errors are used to evaluate the performance, as is common in evaluating fuzzers [38].

[1] https://github.com/moudemans/GFuzz

Each benchmark is run for ten minutes as initial experiments have shown the coverage and error detection stagnate after this time. The performance of PGFuzz and the random fuzzer are compared to the performance of the graph generators, GMark and PGMark. The coverage reports are used to determine which branching is not reached by our proposed methods. The errors detected are simplified to their stack trace, only including the traces that contain the application under test. Then, the performance of each mutation is determined by analyzing the contribution to the increase in coverage and the found errors [62]. Lastly, we run the methods on a benchmark with more nested branching to see how PGFuzz scales with increased depth complexity.

JQF is used to run the fuzz loop and apply instrumentation to the benchmark programs [42]. It is a coverage-guided fuzz testing framework in Java and has been used in many other works with structure-aware fuzzing [11, 43, 63]. A test function can be annotated to run an application using this framework, calling the methods undergoing tests. If the graph format of PGFuzz is not used, the translation function between the two formats must be defined. Following recommendations from previous work to account for the random nature of fuzz testing [32], we have repeated experiments five times. The seeds are generated from the same graph generator, providing new inputs during execution. The generators used are GMark [4] and its extension PGMark[2] for property graphs. We extended PGMark to generate edge properties besides only node properties.

The results were collected from a machine running Windows 11, which has an Intel(R) Core(TM) i7-13700KF 3.40 GHz CPU and 32GB of memory available.

4.2 Benchmarks

It is recommended to have similar benchmarks with other works [32], but as far as we know, no other works have focused on coverage-guided testing of graph processing applications. Benchmarks from different research in the field of graph processing did not fit the criteria, as there was little branching that was not already covered by a trivial input.

The benchmarks used for the evaluation are collected through an extensive search in publicly available repositories (i.e., GitHub and GitLab) and academic works. The search considered repositories with a graph database backend or applications with graph-like structures in memory. The repositories that fit the criteria had code fragments where branching was determined by the graph state extracted. Applications written in Java have been prioritized, as the JQF only supports Java applications, and this requires the least amount of extra steps to fit it in a benchmark. Code bases written in another language have been copied to Java.

In table 4.1, the benchmarks used in the evaluation are shown. Some of the benchmark programs are collected from the same repository but are in distinct benchmarks as they have different functionality and schemas. Most benchmarks in the suite use Neo4J, which

[2] https://github.com/ThomHurks/pgMark

Code	Name	Framework	Language	Characteristics
P1	Medical	Neo4J	Python	Cardinality
P2	Transportation	Neo4J	Python	Properties and node labels
P3	Citation	JanusGraph	Java	Edge properties
P4	Citation Network	JanusGraph	Java	Relationships
P5	Pangenomic	Neo4J	Python	Property values
P6	Pheno4JOut	Neo4J	Java	Property keys and types
P7	Pheno4J	Neo4J	Java	Property types
P8	PanTool1	Neo4J	Java	Property values
P9	PanTool2	Neo4J	Java	Unstructured dependencies

Table 4.1: Benchmark details

is one of the most used graph database engines. Table 4.1 also shows the characteristics, which will be explained later in this section. We tried to minimize the number of overlapping characteristics and maximize the diversity of these characteristics to better represent the graph search space.

P1 Medical - This benchmark has been collected from Openstudybuilder, which is an open-source solution for clinical study evaluations [3]. The benchmark updates a previous node's relationships to a new node, using the node label to make the selection. The selected nodes require the changing relationships to follow a one-to-one cardinality constraint.

P2 Transportation - This benchmark is collected from academic work to assess the impact of demand-responsive services on public transit accessibility [21]. The operations performed on the graph data are filtering, counting, and selection, which assume a specific graph structure to be followed. The branching logic of this example relies on the number of edges between two nodes and the properties on these edges.

P3 Citation - Citegraph is an open-source online visualizer that was initially built as a demo of JanusGraph, an open-source graph database[4]. The application communicates with the graph DB backend to load papers, authors, and citations into a model. The code fragment relies on nodes, edges, and properties with specific labels and typing.

P4 Citation Network - This benchmark was collected from the same repository as the P3 Citation. This segment finds its n-hop references for a given paper. It relies on specific relationships to be connected with at least n connected components.

[3] https://gitlab.com/Novo-Nordisk/nn-public/openstudybuilder/project-description
[4] https://www.citegraph.io/

P5 Pangenomic - Pangenomic is an academic work that provides a data-centric pipeline capable of operating on a unified graph dataset consisting of multiple pangenome graphs[5]. It is based on the PPanGGOLiN framework[6] and uses Neo4J for complex analysis. The benchmark loads properties into a model and checks for connected nodes with similar property values.

P6 Pheno4JOut - Pheno4J is a repository of published research [39] that can be used to export and import data to the Pheno4J database using Neo4J. The code fragment in this benchmark takes a node and uses its labels and types to set up an export schema. The property labels on the nodes and the property types determine the branching.

P7 Pheno4J - This benchmark was collected from the same repository as P6 Pheno4JOut. It loads the graph data into a model. The code fragment used in the benchmark has branching determined by the edge relationships being present and properties being defined.

P8 PanTool1 - PanTool is an extensive code base[7] with thousands of lines of code and multiple associated publication [1, 28, 29, 49, 50]. It is a Pangenomic toolkit for the comparative analysis of a large number of genomes. The property values determine the branching in the benchmark.

P9 PanTool2 - Pantool2 is from the same repository as P8 Pantool1. Similarly to P8, it has branching, which depends on property values. Additionally, the property values require an implicit relationship with other property values in other graph elements. A node can have a property value 'count', which indicates how many other nodes there should be and which values should be in the properties.

4.3 Results

This section will present the collected results to answer the questions formulated at the beginning of Section 4.

4.3.1 Q1. How effective is PGFuzz in increasing branch coverage compared to existing graph generators?

In Figure 4.1, the average coverage per trial is shown for PGFuzz, Random, and the graph generator. The performance of a method is measured by the total coverage in the number of trials, taking both the total coverage and the coverage at each step in the trial into consideration. P4 is not shown in the coverage results, as testing has shown all branches are covered within the first few trials for every test method.

[5] https://github.com/jpjarnoux/PanGraph-DB?tab=readme-ov-file
[6] https://github.com/labgem/PPanGGOLiN
[7] https://git.wur.nl/bioinformatics/pantools/

The results from Figure 4.1 show that the Random approach significantly increases the coverage found over the graph generator baselines in 75% of the benchmarks. PGFuzz shows even better results, reaching more coverage in less time in each benchmark program. On average, PGFuzz can increase the coverage by 23%, with a maximum increase of 57%.

Even though outperforming the other methods, PGFuzz cannot consistently get 100% coverage for each benchmark. P1 has an unreachable condition in a private method, as it has already been checked and filtered in an earlier branch. In P3, PGFuzz has been able to reach 100% coverage, but it seems it can not cover each branch consistently in the given amount of trials. The branches not reached consistently require a specific node to exist and be mutated, making it possible but unlikely to be reached. Lastly, in P9, PGFuzz cannot reach 100% as it requires sequential IDs in node properties, which is highly improbable in both the generated and mutated inputs.

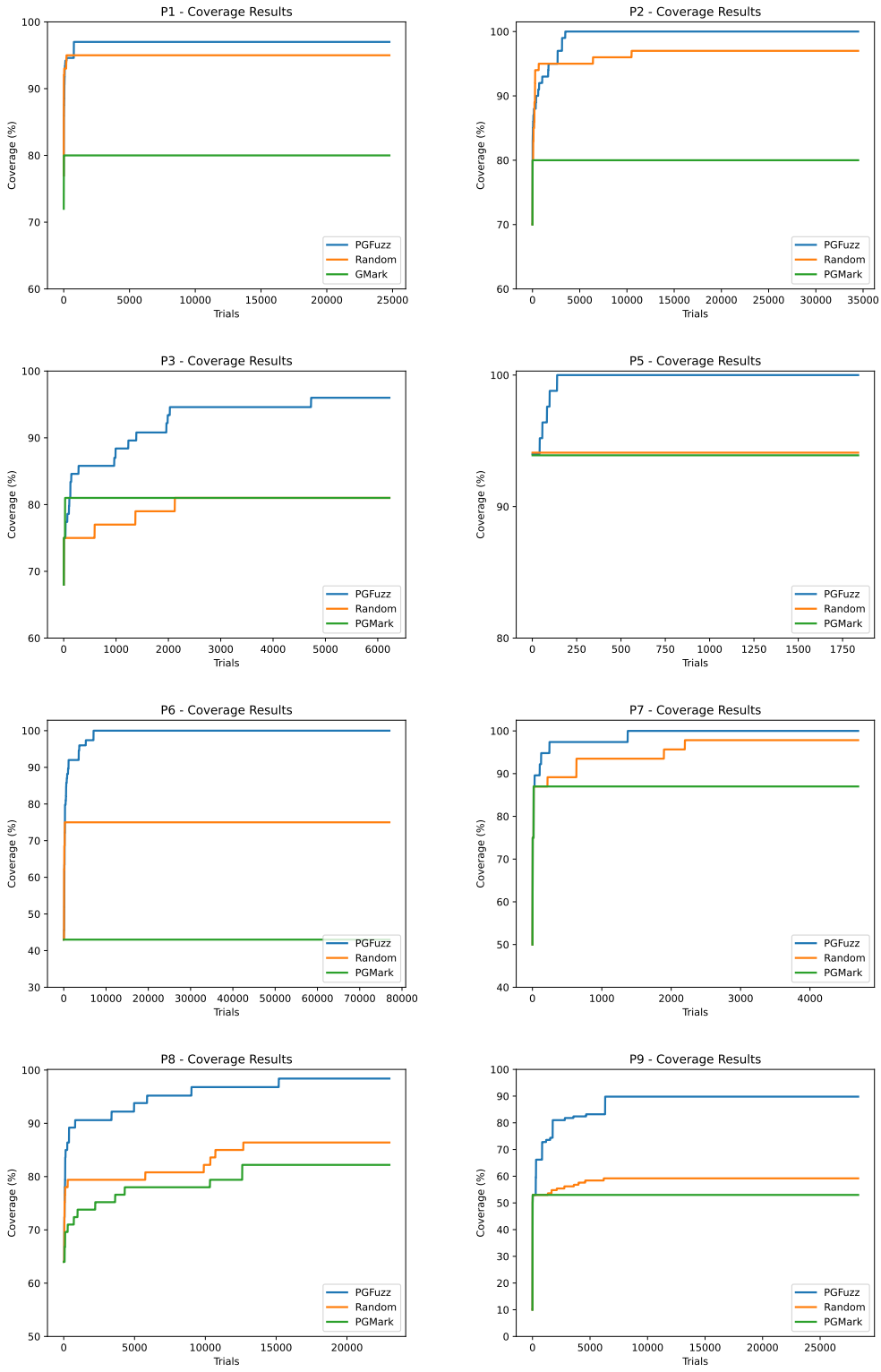


Figure 4.1: Benchmark coverage results

4.3.2 Q2. How effective is PGFuzz in finding unique errors?

Table 4.2 shows the total unique errors for Random and PGFuzz. The graph generators are not included in the table as they only generate graphs that follow the schema and do not produce errors. There are three types of errors found by the PGFuzz and Random in the benchmarks:

1. Invalid data types
2. Null pointers
3. Array index out of bounds

The invalid data types occur the most, 23 times, as the benchmark tries to parse integers, booleans, or other types while the property value is no longer of that type. Null pointers have occurred five times and are triggered by certain graph elements no longer being present. Lastly, the array index out of bounds occurred three times and was caused by removing graph elements or splitting a string that no longer contained a specific character.

PGFuzz can find more unique errors than the Random method in two benchmarks, P1 and P4. These are caused by removed graph elements, which the random method can not achieve. In two benchmarks, P3 and P7, the Random method finds more unique errors. PGFuzz could have found these errors with more time, but the Random approach finds them in fewer trials. This difference could be explained by the Random method having a strong bias towards changing property values with a high chance of making it a string value instead of a number, boolean, or other expected data type. PGFuzz has many other mutations, making the change property type mutations less likely to occur.

PGFuzz can find both logical errors and semantic errors. The logical errors are caused mainly by changing the graph elements, and the semantic errors are caused by changing graph elements and property types. The Random method finds the type errors by changing the property value to a string.

4.3.3 Q3. How much does each mutation mechanism contribute to the increased coverage?

Figure 4.2 shows the relative increase in coverage compared to the overall coverage increase per each mutation category. Each mutation category has been run independently of the others, only allowing one mutation category at a time.

Changing graph elements (C3) impacts the coverage increase most. It has the largest coverage increase contribution in 7 of the 8 benchmarks.

Each mutation contributes to new coverage, as each mutation category appears in at least one of the benchmarks. A single mutation is responsible for the total coverage increase in three benchmarks (P3, P5, and P7). In the other other benchmarks, a combination of

Benchmark	Random	**Error**	PGFuzz	Error
P1 - Medical	0	n.a.	2	Nullpointer
P2 - Transportation	5	Number format exception	5	Number format exception Array index out of bound
P3 - Citation	13	Number format exception	10	Number format exception
P4 - CitationNetwork	0	n.a.	1	Nullpointer
P5 - PanGenomic	2	Nullpointer	2	Nullpointer
P6 - PhenoOut	0	n.a.	0	n.a.
P7 - Pheno4j	9	Number format exception	4	Index out of bound Number format exception
P8 - PanTool1	4	Index out of bound Number format exception	4	Index out of bound Number format exception
P9 - PanTool2	3	Index out of bound Number format exception	3	Index out of bound Number format exception

Table 4.2: Unique errors found per method

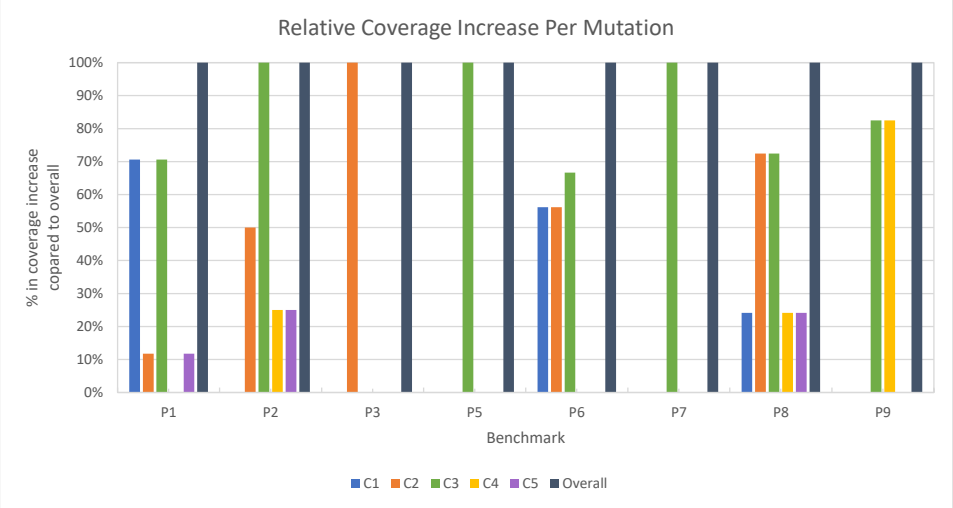


Figure 4.2: Coverage increase contribution per mutation category

mutations is needed to reach the total coverage increase. The mutations appear to overlap, as for some of the benchmarks, the summation of each mutation category increase exceeds 100%. This is expected, as certain mutations can have similar results depending on benchmark logic. An example is Changing graph elements (C3), which can be seen as Removing a graph element (C2) and adding a new Graph element (C1).

4.3.4 Q4. Which mutations are most effective in finding unique errors?

Figure 4.3 shows how many unique errors of each type have been found per mutation category. Similar to the coverage report, the mutation categories have been run independently. The error codes have been labeled according to how they are presented in Section 4.3.2; E1

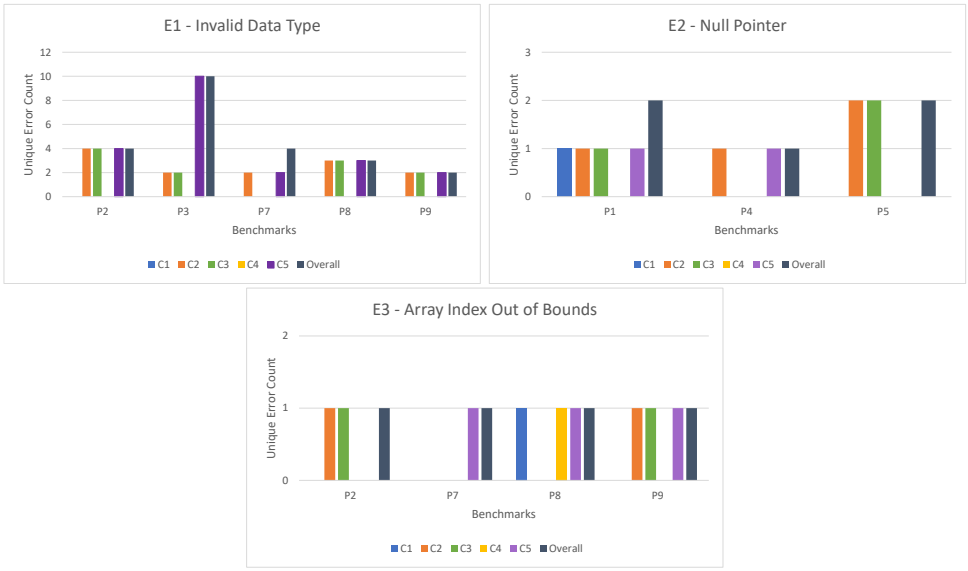


Figure 4.3: Mutation contribution per error type

is an Invalid data type, E2 is a Null pointer, and E3 is an Array index out of bounds.

Breaking graph constraints (C5) appears to be most effective in finding unique errors, detecting the most unique errors in 10 of the 12 benchmarks. Changing data values (C4) contributes the least, only detecting one error in P8.

Each mutation contributes to detecting unique errors, as each mutation category appears in at least one of the benchmarks. Similar to the coverage increase, some mutation categories overlap with others, as the summation of unique errors exceeds the total number of unique errors detected.

The following conclusions can be made from the contribution of each mutation category per error type:

- Invalid data type (E1) are caused by Breaking constraint mutations (C5), Removing graph elements (C2) or Changing graph elements (C3).
- Null pointers (E2) are primarily caused by Removing graph elements (C2), Changing graph elements (C3), and Breaking constraint mutations (C5). Adding a New graph element (C1) rarely causes the error, and Changing the property value (C4) does not trigger the error.
- any of the mutations can cause item Array index out-of-bound error (E3) and is primarily caused by Breaking constraint mutations (C5).

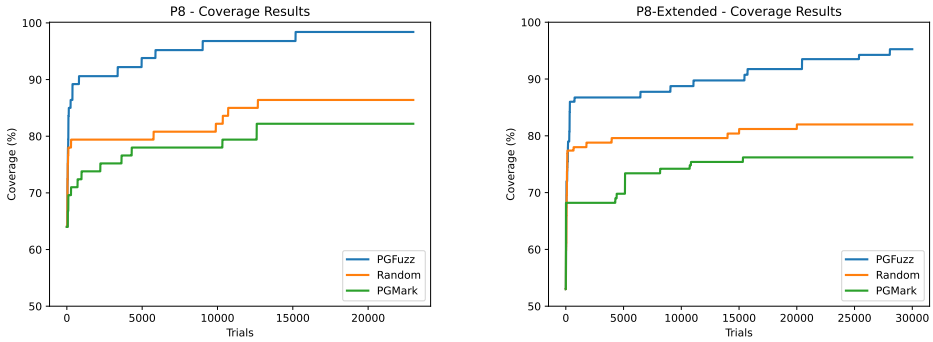


Figure 4.4: Coverage results for AST depth 4 (left) and AST depth 8 (right)

4.3.5 Q5. How does the performance of PGFuzz scale with different application complexities?

The number of nested conditionals can quantify the complexity of a benchmark. These conditionals can be modeled in an abstract syntax tree (AST) where each condition is a node with two edges, one edge for the true path and one for the false path. Each nested branching condition adds 1 to the depth, making it more complex as the input must meet more conditions to reach that depth. To assess PGFuzz's performance under various application complexities, we expanded the existing P8-PanTool1 benchmark by introducing additional conditionals. These new conditions were derived from similar ones in the original application and other benchmarks. The original AST of P8 had a maximum depth of 4. Adding four nested conditionals doubled the original depth, reaching a maximum depth of 8.

Figure 4.4 shows the coverage results for P8 and the extended benchmark. The performance of a method is measured by the total coverage in the number of trials, similar to the analysis in 4.3.1.

PGFuzz is able to cover all additional branching in the extended benchmark. It does reach 100% coverage but does not consistently reach it in the given number of trials. The time it takes PGFuzz to find the additional branching appears to scale with the increased complexity. This is also reflected in the Random method, which requires more trials to cover all reachable branches. PGMark has similar performances but more variance across the experiments.

4.4 Case Study: Cycle detection and Compound Mutations

Cycle detection is a common task within graph processing [6, 31, 46]. It is used across different fields like fraud detection [26, 54] and is part of another well-studied task, pattern counting [14].

We ran PGFuzz on a bread-first search cycle detection algorithm. The algorithm starts from a random node and traverses the neighboring nodes. When an already visited node is reached, the cycle size is checked, and the cycle is returned. If there is no cycle, the algorithm finishes after visiting each node. Any cycle-detecting algorithm for a fixed cycle size would work as long as there is an unreached branch that is only executed when a proper-sized cycle is detected. Initial testing showed that PGFuzz could not make a large enough change to trigger new coverage. If a new node were added, even though it would be one step closer to the cycle size, it would be discarded as now new coverage was achieved.

To solve this problem, we extended the set of mutations with a compound mutation, a mutation that applies existing mutations on a larger scale. The compound mutation is an extension of Add node (M1), now adding multiple nodes to the graph in a single mutation. Whether a graph contains a cycle of size S is influenced by the size of the graph and the density. A graph with more nodes and edges is more likely to have cycles or be close to a cycle (e.g., a cycle is created by adding a single node or edge). We tested the methods on five different graph sizes to gauge the scalability of PGFuzz and its extension with support for compound mutations. The graphs are generated using GMark.

In Figure 4.5, the performance of PGFuzz with the compound mutation can be seen in generating a graph with a cycle of size 10. The compound mutations give a significant advantage when the graph is relatively small. Without the compound mutation, PGFuzz cannot create a cycle and reach new coverage in a graph with size 100. The medium-sized graphs (|N|=200. |N|=500) PGFuzz with the compound mutation enabled can generate the graph in fewer trials. PGFuzz can create the cycle with these sizes, though not consistently. With the larger graphs (|N|=100, |N|=1000), the performance is similar between PGFuzz with and without compound mutations. The graphs are likely close to a cycle, which PGFuzz can create by a single mutation.

This case study shows compound mutations (i.e., mutation combination or repeating mutations) might be useful for applications where:

- Branching depends on large changes to the graph, not achieved by the single mutations of PGFuzz.
- The graph input size needs to be kept as small as possible. For example, for performance or complexity considerations.
- For applications requiring specific patterns (e.g., cycle, triangle).

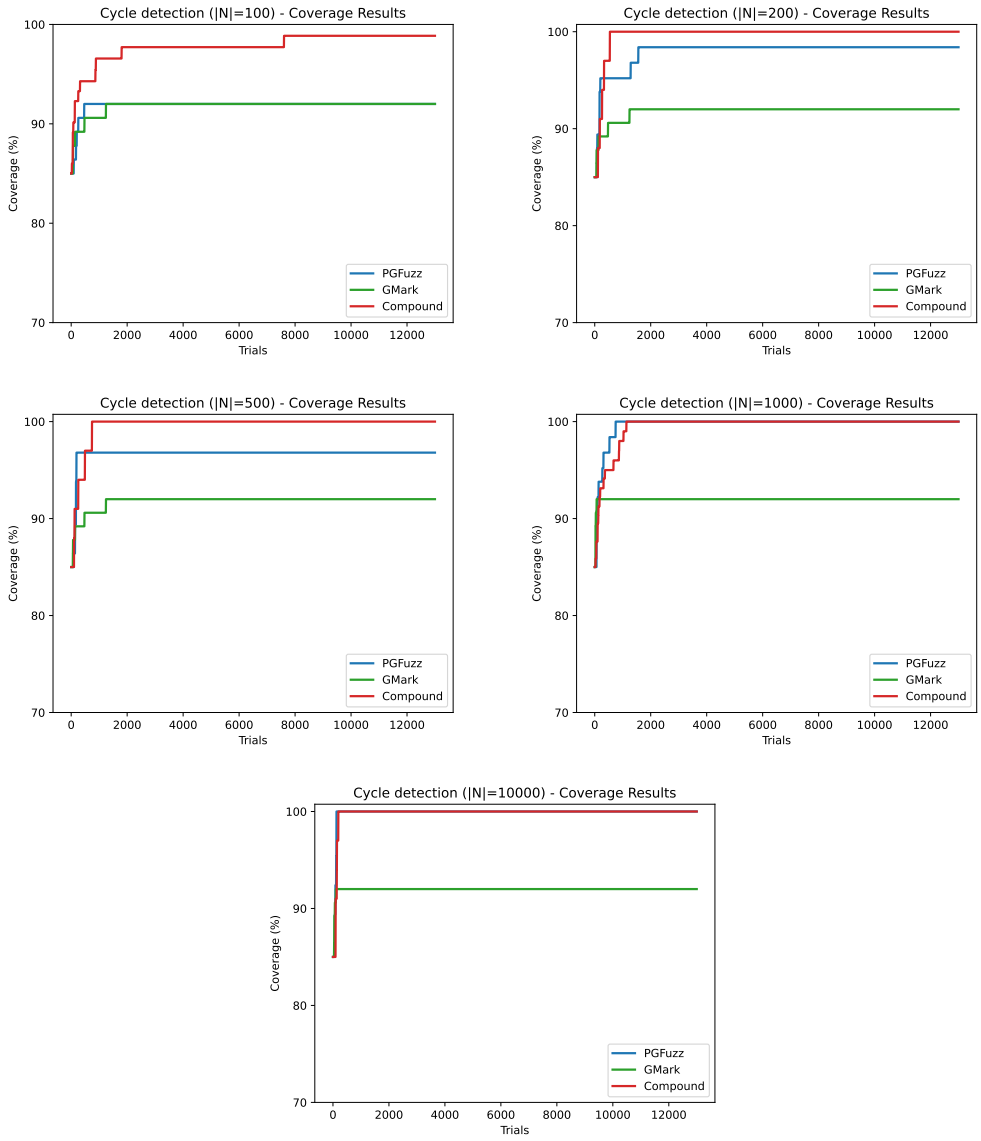


Figure 4.5: Coverage results detecting cycles with node count |N|

We also ran the compound variant on the benchmark suite used in the previous section. The results are added in Appendix A and show no performance difference. Compound mutations do not affect performance for large graphs in cycle detection and applications with fewer changes needed to satisfy new branching.

4.5 Evaluation Summary

This section will summarize the answers to the questions used to evaluate PGFuzz.

Q1. How effective is PGFuzz in increasing branch coverage compared to existing graph generators?
PGFuzz shows an increase in the branch coverage in each of the benchmarks. On average, PGFuzz increases the absolute coverage by 23%, with a maximum increase of 57%.

Q2. How effective is PGFuzz in finding unique errors?
PGFuzz can detect both semantic and logical errors. The types of errors found are invalid data types, null pointers, and array index out-of-bounds.

Q3. How much does each mutation mechanism contribute to the increased coverage?
Changing graph elements (C3) impacts the coverage increase most. It has the most significant coverage increase contribution in 7 of the 8 benchmarks

Q3. Which mutations are most effective in finding unique errors?
Breaking graph constraints (C5) appears to be most effective in finding unique errors, detecting the most unique errors in 10 of the 12 benchmarks. Invalid data types are caused by Breaking constraint mutations (C5), Removing graph elements (C2), or Changing graph elements (C3). The Null pointers are primarily caused by Removing graph elements (C2), Changing graph elements (C3), and Breaking constraint mutations (C5). The Array index out-of-bound error has been primarily caused by Breaking constraint mutations (C5)

Q4. How does the performance of PGFuzz scale with different application complexities? The time it takes PGFuzz to cover branching scales with the application's nested branching depth. Even with increasing branching depth, PGFuzz can still cover all branching.

Chapter 5

Related Work

This chapter discusses existing work related to the main components PGFuzz uses. It is divided into four sections: first, generating (valid) states relevant to seed inputs and new inputs for PGFuzz; second, testing applications with a non-graph database backend; third, fuzz testing; and finally, testing graph database engines.

5.1 Graph generation

Generating graph states can be achieved by using a graph generator. Existing graph generators can be categorized into five different groups [36]: traditional, community structures, community structure and node attribute, large scale, and neural network-based. The traditional graph generators focus on edge density and node distributions. Generators for community services have a similar approach to the traditional but also consider the topological structures within communities. The generators for community structures and node attributes can generate node attributes in the graph and generate edges between nodes based on similarities. The neural network-based graph generators try to produce real-world graphs for neural networks, following either a sample input or a graph characteristic. As the name suggests, large-scale graph generators can efficiently generate large graphs using schemas that support node labels and edge predicates.

GenCat [36] is a graph generator that stands out for its capability to generate graphs with user-specified node degrees, relationship control proportion for each node to classes, and attribute distributions. The generator can produce these complex graphs in linear time, scaling with the number of edges. Gencat also solves both core/border and homophily/heterophily phenomena with this approach (i.e., certain node groups are highly interconnected or very sparse). GenCat has been further developed and is used in more specialized research for GNN's [35]. Both works show that their attribute generation is focused on creating attributes useful for neural networks and other language models. However, the generator cannot handle different property types or cardinality constraints.

GMark is a generator that generates both graph instances as query workloads from a schema

[4]. It is a platform-independent tool where the user can define the schema node, edges, overall graph properties, cardinality constraints, and distributions. GMark has shown to be a novel generation tool that allows for quality graphs to be generated. While the approach performs well for generating graphs, it does not provide the full functionality of a property graph, as node and edge property values can not be generated. PGMark is an extension of GMark and is used in other work [19]. PGMark has similar functionality but also provides functionality to generate node property values with specific ranges or constraints.

It shows that the kind of graph generator used depends highly on the domain. Most generators utilize some form of graph schema, highlighting the complexity of each structure. PGFuzz can be used alongside any of these graph generators, taking advantage of their diversity and functionality.

5.2 Testing invalid database states

Multiple works have shown testing methods that avoid or at least minimize the number of invalid database states [12, 16, 17, 45]. Their approaches vary, from having user input to predefined schemas or specialized mutations.

For a relational database, for example, you can use the conceptual model of the database to generate states to test all ER model constructs and the database constraints [12]. AGENDA uses sample value data for attributes that can be written to the database[16]. Instead of an automatic mutation or guidance, the user specifies the behavior of test cases interactively. The approach does not consider invalid database states, which might be less of a problem since mutations are heavily user-guided and mutation values are pre-defined. Having data outside the allowed schema is for future consideration [13]. An extension of AGENDA has been proposed, which translates constraints that the DBMS does not enforce into simpler constraints [17]. They acknowledge that schema-breaking states could be interesting, but they filter those by following a static analysis to determine which queries return valid results.

SynDB is a method to test database applications where the original code is rewritten to a synthesized database where they can apply Pex [45]. This engine uses a constraint solver to generate new inputs. The database constraints are also transformed to validate the database state. The first check is done within the synthesized database, and the second filtering is done by Pex, which filters out invalid inputs. Invalid database states are, therefore, not considered and not evaluated.

Unlike the other examples, DBGriller stands out as it tries to increase coverage and find unique bugs, similar to PGFuzz [62]. The challenge is to generate valid database states effectively [62] to pass validation checks but still target exploration and logic bugs in the application. DBGriller has been specifically designed for relational databases and has inspired the work for PGFuzz.

5.3 Fuzz Testing

Fuzz testing is a popular and effective automated testing method across multiple fields [34, 66]. The challenge is often to have a good input generator and mutator. The most straightforward method is to use random bit- or byte mutation, but often, specific mutations are designed for more complex domains. For example, multiple works proposed specialized methods for big data analytics. DepFuzz keeps track of data dependencies from the input [24]. It tracks which code segments operate on which datasets, rows, and columns. DepFuzz generates test data that should reach hard-to-reach regions of the application code by tracking dataflow operators and semantics using taint analysis. NaturalFuzz aims to create as ”natural” or ”realistic” inputs as possible [25]. The natural mutations are generated by selecting which columns to mutate on and combining similar data points across different rows and columns to construct new realistic synthetic data.

Applying fuzz testing with coverage feedback to test relational database management systems has been covered by Squirrel[65]. They provide type-based mutations that generate syntactically correct queries. They analyze their custom query representation for dependencies between arguments to avoid semantic errors. The mutations are specifically designed for SQL queries and, thus, not applicable to graph structures.

There is little work on mutating graph structures. GraphFuzz, coming close, represents every test case as a dataflow graph where vertices are functions and edges are object dependencies (hence GraphFuzz). GraphFuzz models sequences of executed functions as a dataflow graph, thus enabling it to perform graph-based mutations at the data and execution trace level. The mutations are specifically designed to change the order of the function calls. The re-ordered graphs are then completed so that dependencies are met, like having the correct final function call at the end of the graph. This mutation would be less effective on property graphs as certain elements are not changed, and the mutations do not consider any semantic constraints.

5.4 Testing Graph Database Engines

Grand is an approach for finding logic bugs in graph databases that adopt Gremlin in the query language [64]. It creates semantically equivalent databases for multiple graph database management systems and compares the results from the engines. The main contribution is generating syntactically correct and valid Gremlin queries with a high probability of returning non-empty results. It does so by using a model-based query generation approach, which is specifically designed for Gremlin. The second problem being tackled is uniforming the query results across different graph database engines.

GDsmith ensures that each randomly generated query satisfies the semantic requirements [22]. it is a graph-guided generation of complex pattern combinations, the graph being generated from the database. They mutate the extracted patterns and use them for query generation, data-guided generation of complex conditions, and static query analysis during

the query generation process. The work is applied to Cypher and touches upon Neo4J, with the goal of finding the wrong results and bugs. GDSmith generates random property graph schemas from which random graph databases are made.

GDBMeter tests graph database systems by partitioning queries [30]. Multiple queries are derived from a query whose results can be combined to reconstruct the given query result. It demonstrates how the Query Partitioning test oracle, particularly Ternary Logic Partitioning, can be applied to graph database systems to find logic bugs.

Dinkel proposes a technique for generating complicated and valid Cypher queries with complicated dataflow and data dependencies [27]. They do this by using the query context and the abstract graph summary. Apply state manipulation to modify the information. It is a novel approach to Cypher query generation, enabling fuzzing of graph database engines using complex queries containing intertwined, data-dependent clauses.

The above works primarily focus on testing graph database engines by custom queries and applying differential testing to find bugs. For the applications to be tested with PG-Fuzz, there is usually no other application against which the results can be compared, which makes it less useful. Additionally, the queries generated by their methods do not capture any application constraints and miss any semantics many applications require.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this paper, we have presented PGFuzz, a coverage-guided, schema-aware fuzzer for graph processing applications. To our knowledge, PGFuzz is the first method to automatically test applications with a graph database backend. PGFuzz has been specifically designed for property graphs and can be used on top of any graph generator. The application is tested using a fuzz loop to pass inputs to the instrumented application, process the coverage feedback, and mutate graph states to produce new inputs. The mutations of PGFuzz can add, remove, or change graph elements and change property values. Additionally, specific mutations target the graph constraints like cardinality, key types, and property types.

We implemented a Random mutator to compare the performance of PGFuzz to a random mutation approach. The Random mutator only mutates element labels and property values, providing the same mutation power as the random byte mutations and significantly reducing the number of corrupted states.

The evaluation shows that PGFuzz achieves more coverage and unique errors in less time than the graph generators in each benchmark we tested. The graph generators already achieved high coverage but could not reach every branch. The Random method reaches a higher coverage but cannot reach every branch due to the limited mutation power. PGFuzz shows an increase in the branch coverage in each of the benchmarks. On average, PGFuzz increases the absolute coverage by 23%, with a maximum increase of 57%.

Both PGFuzz as the Random method to detect three types of errors: Invalid data types, Null pointers, and Array index out of bounds. These errors are primarily caused by Removing graph elements, changing graph elements, or breaking constraint mutations.

Even with more complex branching, PGFuzz can still reach every branch. The number of trials it takes to cover extra branching appears to scale with the program's branching depth, as expected.

6.2 Discussion

In our evaluation, we used graph schemas that were either collected from the benchmark source or derived from the logic found in the code. We did not consider invalid schemas or seeds to be passed to PGFuzz. No validation checks are built for either the seed or schema, and PGFuzz is not designed to validate graph states. We can assume the performance of PGFuzz might change significantly if one of these is incorrect, as is recognized in other work [8].

We used two recent works to design the mutations for PGFuzz. PG-Key and PG-Schema describe constraints and schemas that should be available for any future standardized graph querying language [2, 3]. We tried to use similar constraints in this work but could not get a similar expressiveness of the scope of a key constraint. In PG-Key, a key type can be defined on any unary result from a graph query. PGFuzz only allows for the key types to be defined per property.

6.3 Future work

PGFuzz and its mutations can be useful in the field of graph pattern generation and detection. During development, we experimented with putting patterns in the graph schema and implementing mutations, but we abandoned it due to time restrictions and a lack of useful applications to test. Our case study shows that PGFuzz with compound mutations is interesting for cycles, and thus, it might also be useful for other kinds of pattern tasks.

Not all mutations require the graph schema to work. Additionally, some information currently collected from the schema could be derived from the graph metadata, and other works describe how to infer a schema from a graph [9, 33]. Future work might examine whether such an automated schema can achieve similar performances as with the manually created schemas in this work. Requiring the schema would reduce possible points of failure and manual work from developers.

Lastly, further extending the mutations of PGFuzz and testing the fuzzer on a more extensive set of benchmarks could be interesting. Finding example programs was challenging as the criteria for useful programs did not occur often in the applications we found. Eventually, the most applicable applications had implicit constraints on the graph structures and where the data was modeled in memory. The graph community is working towards a standardized query language that might introduce new kinds of constraints, which can be modeled into mutations.

Bibliography

[1] Siavash Sheikhizadeh Anari, Dick de Ridder, M Eric Schranz, and Sandra Smit. Pangenomic read mapping. *bioRxiv*, page 813634, 2019.

[2] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W Hare, Jan Hidders, Victor E Lee, Bei Li, Leonid Libkin, Wim Martens, et al. Pg-keys: Keys for property graphs. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2423–2436, 2021.

[3] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. Pg-schema: Schemas for property graphs. *Proceedings of the ACM on Management of Data*, 1:1–25, 6 2023. doi: 10.1145/3589778.

[4] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George HL Fletcher, Aurélien Lemay, and Nicky Advokaat. gmark: Schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, 29(4):856–869, 2016.

[5] Omar Batarfi, Radwa El Shawi, Ayman G Fayoumi, Reza Nouri, Seyed-Mehdi-Reza Beheshti, Ahmed Barnawi, and Sherif Sakr. Large scale graph processing systems: survey and an experimental evaluation. *Cluster Computing*, 18:1189–1213, 2015.

[6] Sayan Bhattacharya and Janardhan Kulkarni. An improved algorithm for incremental cycle detection and topological ordering in sparse graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2509–2521. SIAM, 2020.

[7] Monica Bianchini, Marco Gori, and Franco Scarselli. Inside pagerank. *ACM Transactions on Internet Technology (TOIT)*, 5(1):92–128, 2005.

[8] Raquel Blanco and Javier Tuya. A test model for graph database applications: an mda-based approach. In *Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation*, pages 8–15, 2015.

[9] Angela Bonifati, Stefania Dumbrava, Emile Martinez, Fatemeh Ghasemi, Malo Jaffré, Pacôme Luton, and Thomas Pickles. Discopg: property graph schema discovery and exploration. *Proceedings of the VLDB Endowment*, 15(12):3654–3657, 2022.

[10] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. *Querying graphs*. Springer Nature, 2022. pg. 3.

[11] Sicong Cao, Biao He, Xiaobing Sun, Yu Ouyang, Chao Zhang, Xiaoxue Wu, Ting Su, Lili Bo, Bin Li, Chuanlei Ma, et al. Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2726–2743. IEEE, 2023.

[12] Man-Yee Chan and Shing-Chi Cheung. Testing database applications with sql semantics. In *CODAS*, volume 99, pages 363–374. Citeseer, 1999.

[13] David Chays, Yuetang Deng, Phyllis G Frankl, Saikat Dan, Filippos I Vokolos, and Elaine J Weyuker. An agenda for testing relational database applications. *Software Testing, verification and reliability*, 14(1):17–44, 2004.

[14] Justin Y Chen, Talya Eden, Piotr Indyk, Honghao Lin, Shyam Narayanan, Ronitt Rubinfeld, Sandeep Silwal, Tal Wagner, David P Woodruff, and Michael Zhang. Triangle and four cycle counting with predictions in graph streams. *arXiv preprint arXiv:2203.09572*, 2022.

[15] H. Conrad. Cunningham, P. (Paul) Ruth, Nicholas A. Kraft, and Association for Computing Machinery. *A Comparison of a Graph Database and a Relational Database*. Association for Computing Machinery, 2010. ISBN 9781450300643.

[16] Yuetang Deng, Phyllis Frankl, and Jiong Wang. Testing web database applications. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

[17] Yuetang Deng, Phyllis Frankl, and David Chays. Testing database transactions with agenda. In *Proceedings of the 27th international conference on Software engineering*, pages 78–87, 2005.

[18] Fabio Duarte. Amount of data created daily (2024), Dec 2023. URL `https://explodingtopics.com/blog/data-generated-per-day#category`.

[19] Zaiwen Feng, Wolfgang Mayer, Keqing He, Selasi Kwashie, Markus Stumptner, Georg Grossmann, Rong Peng, and Wangyu Huang. A schema-driven synthetic knowledge graph generation approach with extended graph differential dependencies (gdd x s). *IEEE Access*, 9:5609–5639, 2020.

[20] José Guia, Valéria Gonçalves Soares, and Jorge Bernardino. Graph databases: Neo4j analysis. In *ICEIS (1)*, pages 351–356, 2017.

[21] Cathia Le Hasif, Andrea Araldo, Stefania Dumbrava, and Dimitri Watel. A graph-database approach to assess the impact of demand-responsive services on public transit accessibility. In *Proceedings of the 15th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, pages 1–4, 2022.

[22] Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, Wenpin Jiao, and Tao Xie. Gdsmith: Detecting bugs in cypher graph database engines. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 163–174, 2023.

[23] Xinyue Huang, Anmin Zhou, Peng Jia, Luping Liu, and Liang Liu. Fuzzing the android applications with http/https network data. *IEEE Access*, 7:59951–59962, 2019.

[24] 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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1050–1061, 2023.

[25] Ahmad Humayun, Yaoxuan Wu, Miryung Kim, and Muhammad Ali Gulzar. Naturalfuzz: Natural input generation for big data analytics. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1592–1603. IEEE, 2023.

[26] Paul Irofti, Andrei Patrascu, and Andra Baltoiu. Quick survey of graph-based fraud detection methods. *arXiv preprint arXiv:1910.11299*, 2019.

[27] Zu-Ming Jiang and Zhendong Su. Dinkel: Fuzzing graph databases with complex and valid cypher queries bachelor's thesis by dominic wüst advanced software technologies lab eth zürich supervised by, 2023.

[28] Eef M Jonkheer, Balázs Brankovics, Ilse M Houwers, Jan M van der Wolf, Peter JM Bonants, Robert AM Vreeburg, Robert Bollema, Jorn R de Haan, Lidija Berke, Sandra Smit, et al. The pectobacterium pangenome, with a focus on pectobacterium brasiliense, shows a robust core and extensive exchange of genes from a shared gene pool. *BMC genomics*, 22:1–18, 2021.

[29] Eef M Jonkheer, Dirk-Jan M van Workum, Siavash Sheikhizadeh Anari, Balázs Brankovics, Jorn R de Haan, Lidija Berke, Theo AJ van der Lee, Dick de Ridder, and Sandra Smit. Pantools v3: functional annotation, classification and phylogenomics. *Bioinformatics*, 38(18):4403–4405, 2022.

[30] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. Testing graph database engines via query partitioning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 140–149, 2023.

[31] Bill Kay, Catherine Schuman, Jade O'connor, Prasanna Date, and Thomas Potok. Neuromorphic graph algorithms: Cycle detection, odd cycle detection, and max flow. In *International Conference on Neuromorphic Systems 2021*, pages 1–7, 2021.

[32] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.

[33] Xue Lei. *Property graph schema extraction*. PhD thesis, Master's thesis, Eindhoven University of Technology, 2021.

[34] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1:1–13, 2018.

[35] Seiji Maekawa, Koki Noda, Yuya Sasaki, et al. Beyond real-world benchmark datasets: An empirical study of node classification with gnns. *Advances in Neural Information Processing Systems*, 35:5562–5574, 2022.

[36] Seiji Maekawa, Yuya Sasaki, George Fletcher, and Makoto Onizuka. Gencat: Generating attributed graphs with controlled relationships between classes, attributes, and topology. *Information Systems*, 115:102195, 2023.

[37] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[38] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 1393–1403, 2021.

[39] Sajid Mughal, Ismail Moghul, Jing Yu, Tristan Clark, David S Gregory, and Nikolas Pontikos. Pheno4j: a gene to phenotype graph database. *Bioinformatics*, 33(20): 3317–3319, 2017.

[40] Neo4j, Jun 2024. URL `https://neo4j.com/graphs4good/covid-19/`.

[41] Neo4j, Jun 2024. URL `https://neo4j.com/use-cases/`.

[42] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 398–401, 2019.

[43] 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.

[44] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. pages 31–42. Association for Computing Machinery, Inc, 7 2019. ISBN 9781450362245. doi: 10.1145/3293882.3330576.

[45] Kai Pan, Xintao Wu, and Tao Xie. Guided test generation for database applications via synthesized database interactions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(2):1–27, 2014.

[46] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 11(12):1876–1888, 2018.

[47] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, MacIej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. The future is big graphs. *Communications of the ACM*, 64:62–71, 9 2021. ISSN 15577317. doi: 10.1145/3434642.

[48] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th international conference on scientific and statistical database management*, pages 1–12, 2013.

[49] Siavash Sheikhizadeh, M Eric Schranz, Mehmet Akdel, Dick de Ridder, and Sandra Smit. Pantools: representation, storage and exploration of pan-genomic data. *Bioinformatics*, 32(17):i487–i493, 2016.

[50] Siavash Sheikhizadeh Anari, Dick de Ridder, M Eric Schranz, and Sandra Smit. Efficient inference of homologs in large eukaryotic pan-proteomes. *BMC bioinformatics*, 19:1–11, 2018.

[51] Solid IT, 2024. URL `https://db-engines.com/en/ranking_trend/graph+dbms`.

[52] Sakshi Srivastava and Anil Kumar Singh. Graph based analysis of panama papers. In *2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, pages 822–827. IEEE, 2018.

[53] Sakshi Srivastava and Anil Kumar Singh. Fraud detection in the distributed graph database. *Cluster Computing*, 26:515–537, 2 2023. ISSN 15737543. doi: 10.1007/s10586-022-03540-3.

[54] Sakshi Srivastava and Anil Kumar Singh. Fraud detection in the distributed graph database. *Cluster Computing*, 26(1):515–537, 2023.

[55] Shixuan Sun and Qiong Luo. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1083–1098, 2020.

[56] Yuanyuan Tian. The world of graph databases from an industry perspective, 2022.

[57] Yuanyuan Tian. The world of graph databases from an industry perspective. *ACM SIGMOD Record*, 51(4):60–67, 2023.

[58] Santiago Timón-Reina, Mariano Rincón, and Rafael Martínez-Tomás. An overview of graph databases and their applications in the biomedical domain, 2021. ISSN 17580463.

[59] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

[60] Ran Wang, Zhengyi Yang, Wenjie Zhang, and Xuemin Lin. An empirical study on recent graph database systems. In *Knowledge Science, Engineering and Management: 13th International Conference, KSEM 2020, Hangzhou, China, August 28–30, 2020, Proceedings, Part I 13*, pages 328–340. Springer, 2020.

[61] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828, 2016.

[62] Cong Yan, Suman Nath, and Shan Lu. Generating test databases for database-backed applications. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2048–2059. IEEE, 2023.

[63] 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*, pages 722–733, 2020.

[64] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. Finding bugs in gremlin-based graph database systems via randomized differential testing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 302–313, 2022.

[65] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 955–970, 2020.

[66] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

Appendix A

Compound Mutations Coverage Results

In figure A.1 the coverage results with compound mutations enabled can be seen. We see that enabling Compound mutations for PGFuzz does not provide a performance difference.

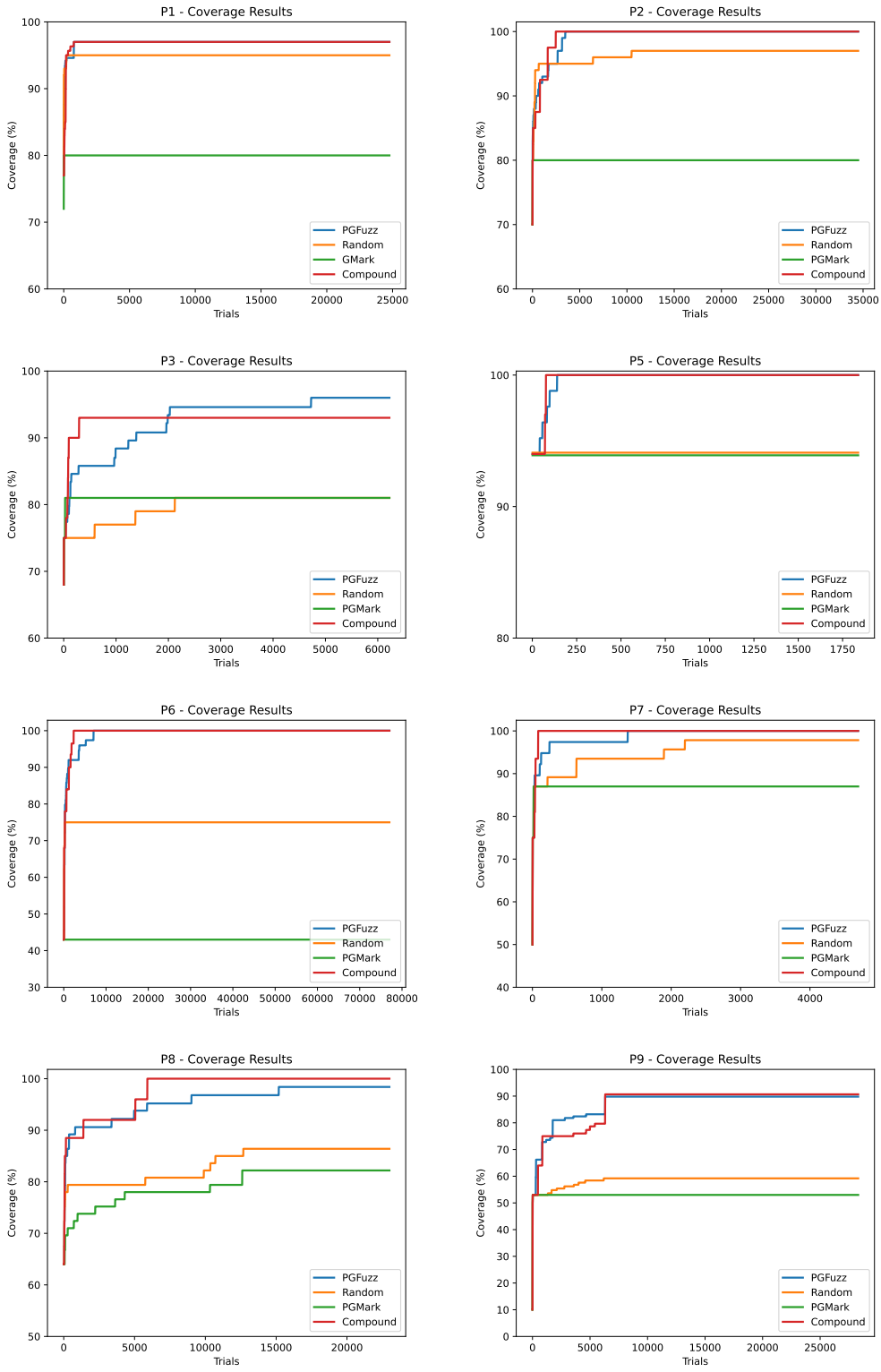


Figure A.1: Coverage results with Compound mutations