Fuzzing Graph Database Applications with Graph Transformations

Stefania Dumbrava[1(✉)], Melchior W. M. Oudemans[2], and Burcu Kulahcioglu Ozkan[2]

[1] ENSIIE & SAMOVAR - Télécom SudParis, Paris, France
stefania.dumbrava@ensiie.fr
[2] Delft University of Technology, Delft, Netherlands

Abstract. Graph databases have surged in popularity, and applications increasingly employ them to store and retrieve interconnected data. However, testing graph database-backed applications has distinctive challenges. Due to the sheer dimension of the graph schema state space, testing applications using naive random graph instances is unlikely to cover a large portion of an application program. We present PGFuzz, a graph transformation-based greybox fuzzer for testing graph database-backed applications, that is, to the best of our knowledge, the first fuzzer to specifically target graph database applications. PGFuzz builds on top of state-of-the-art graph generators and utilizes graph transformations guided by code coverage to produce application test inputs. PGFuzz's graph transformations are schema-aware and support recently introduced graph schema, key, and cardinality constraints. We evaluate PGFuzz on graph database applications that we curate from open-source repositories and show that PGFuzz substantially improves the test coverage of graph database-backed applications compared to the state-of-the-art.

Keywords: graph database applications · graph transformations · graph schema constraints · fuzzing · automated testing

1 Introduction

The popularity of graph database management systems (GDBMSs) has surged in recent years, driven by the increasing need to efficiently store, process, and analyze complex graph data [71]. Due to being custom-built to handle relationship-centric scenarios, GDBMSs have numerous use cases [40,58,70] involving highly connected data, ranging from healthcare [7,14,38,67], finance [15,36,50], to transportation [39,41,65], and various enterprise applications [72].

Prominent examples of commercial GDBMSs include both *native graph* and *multi-model* systems. The former leverage *graph data models*, such as in Neo4j [57], MemGraph [53], or TigerGraph [76], or *triple stores*, as in Amazon Neptune [16], AllegroGraph [8], or BlazeGraph [21]. Multi-model systems mix several models, with the main ones being, for example, *wide-column stores*

J. Endrullis and M. Tichy (Eds.): ICGT 2025, LNCS 15720, pp. 135–156, 2025.
https://doi.org/10.1007/978-3-031-94706-3_7

(JanusGraph [46], DataStax [27], Titan [77], etc.), *key-value stores* (HyperGraphDB [42], DGraph [28], RedisGraph [69], etc.), *document stores* (Azure Cosmos DB [55], ArangoDB [13], OrientDB [61], etc.) or *relational tables* (AgensGraph [20], Db2 Graph [43], Oracle Spatial and Graph [60], etc.). In this work, we target native GDBMSs, represented by the popular Neo4j database, and multi-model ones, represented by JanusGraph, one of the few GDBMSs equipped with not only graph schema, but also advanced integrity constraint mechanisms [11].

The behavior of graph database-backed applications depends on the contents of the input graph database. Therefore, exercising various executions of an application's logic requires the use of different input graph instances, as they can activate different parts of the application logic. Unforeseen input database instances may result in unexpected and erroneous application behaviors.

Ensuring the reliability of graph database applications through testing faces significant challenges in terms of *validity* and *effectiveness*. First, the database instances for the graph database applications must satisfy complicated constraints determined by application semantics, which are nontrivial to synthesize. Second, effectively exploring the application logic requires generating graph database instances that can trigger various parts of the application code.

An intuitive approach to test graph database-backed applications is to stress-test them using graph instance generators, such as the state-of-the-art schema-driven graph generators gMark [18] and pgMark [75]. However, while these schema-driven generators solve the problem of validity, they do not offer effective testing of the applications. They only generate graphs that comply with a given graph data schema, which makes them inherently limited in their coverage of all application behaviors. Although most graph databases do not enforce schema constraints, these are crucial to data quality [22,44], data integration, and data exchange [19,33]. The application programs process the retrieved data and ensure its integrity. As such, they perform operations and substitutions on graph database instances. However, schema-driven graph generators often fail to cover all branches of an application, particularly logic involving application-specific substitution values. Additionally, these generators produce database states independently, without leveraging insights from previously generated states. This limits their ability to cover complex applications with nested branching logic.

Fuzzing [52,82,84] is a popular approach for software testing. Blackbox fuzzing treats the program as a black box, i.e., it does not know the internal structure and branching of the program and generates test inputs randomly, independent of the program logic and the other test inputs. In contrast, white-box fuzzing [35] analyzes the program's internals to create test inputs. This involves examining the program's branching logic and using satisfiability solvers to generate inputs that can trigger specific branches. While blackbox fuzzing may struggle to effectively exercise rare or deep code branches by creating inputs that can navigate less frequent paths, whitebox fuzzing requires complete knowledge of the system being tested and is computationally intensive.

Greybox fuzzing [82] combines both effectiveness and efficiency by utilizing information gathered from the program being tested, such as execution branch coverage, to generate new test executions. A prominent approach is *coverage-guided* greybox fuzzing, which creates new test inputs by modifying previous inputs that trigger new coverage during execution. Testing the application with these modified test cases allows for more effective exploration of low-frequency or deeply nested program paths and helps with the discovery of new paths. Greybox fuzzing has been shown to be an effective automated approach for testing various application domains [52,84]. However, to our knowledge, there are *no fuzzers designed to generate graph inputs for graph database-backed applications.*

Fuzzing at the input level is often insufficient for graph-backed applications, where control flow depends on the structure and content of the database, such as missing properties, relationship counts, or specific subgraph patterns. PGFuzz addresses this by targeting the database state directly to explore such logic. It is the first greybox fuzzer for graph database-backed applications that combines schema-aware and constraint-aware graph transformations with coverage-guided feedback. To ensure validity, PGFuzz builds on top of state-of-the-art gMark [18] and pgMark [75] graph generators and then mutates the generated instances to uncover new application behaviors. To address scalability, PGFuzz operates on an in-memory graph model and simulates database calls through a lightweight API, avoiding the overhead of actual GDBMS operations and enabling fast, efficient fuzzing across large input spaces.

While PGFuzz is the first greybox fuzzer targeting graph database-backed applications, its primary novelty is its use of *schema-aware mutations*, which produce input graphs that *respect schema-defined structure and types* while deliberately *violating key and cardinality constraints* through targeted property mutations. These mutations, informed by the recent PG-Schema [11] language, contrast grammar-based fuzzers—which strictly enforce grammatical constraints—by intentionally generating graph instances that are syntactically valid, according to the schema types, but that violate key and cardinality constraints. This allows PGFuzz to thoroughly test the application logic responsible for maintaining data integrity and schema compliance.

Testing graph database-backed *applications* is an underexplored area, and there are no openly available sets of such benchmark applications. For the empirical evaluation, we examined open-source applications built on top of Neo4j [57], one of the most popular graph databases [73], and JanusGraph [46], one of the few graph databases to enforce schema constraints [11], and compiled a custom benchmark suite for testing database-backed applications. We evaluate PGFuzz against existing graph generators for test generation and show its superior coverage. Our tests revealed several bugs in the application programs that manifest when using certain graph database instances as input states.

In summary, this paper makes the following contributions:

– PGFuzz, the first greybox fuzzer for testing graph database-backed applications that uses schema and constraint-aware graph transformations to generate input graph database instances.

– A benchmark suite collected from real-world graph database-backed applications that leverage heterogeneous graph structures from various domains.
– An empirical analysis evaluating PGFuzz against the gMark and pgMark state-of-the-art schema-driven graph instance generators.

```
void getTransport(Graph g) {
 int transport_count = 0;
 List<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;
}
```

(a) Excerpt from P2 [39].

```
List<PaperResponse> referers =
g.getConnectedNodes
 (paper, "cites", true)
.stream()
.map(v -> {
 Map<String, String> props =
 v.properties;
 return new PaperResponse(
  (String) (v.id + ""),
  (String) props.getOrDefault
   ("title", "N/A"),
  Integer.parseInt(props.getOrDefault
   ("year", "N/A")),
  Integer.parseInt(props.getOrDefault
   ("numOfPaperReferees", "0")),
  Integer.parseInt(props.getOrDefault
   ("numOfPaperReferers", "0")),
  Double.parseDouble(props.getOrDefault
   ("pagerank", "0.0"))); })
```

(b) Excerpt from P3 [3].

Fig. 1. Simplified code excerpts from our benchmark applications.

2 Motivating Examples

We showcase PGFuzz's key challenges with an example involving branching logic based on the input graph schema and database state, and another handling schema-constrained graph instances where the application ensures data integrity.

Example 1 (Transport Graphs). Graph databases are increasingly used to analyze existing public transit networks and to construct novel models that improve accessibility [29,41,65]. In a recent use-case [39], GTFS[1] schedule data has been used to build and process public transportation graph models in the Neo4j graph database. A code snippet from the application is shown in Fig. 1a. Its purpose is to analyze the connectivity of the `centroid` nodes (centers of a tessellated grid map) using demand-responsive transport (edges labeled "DRT") and pedestrian routes (edges labeled "WALK"). Note that the first branching condition depends on these labels, which are part of the typing schema constraint of the underlying graph database. *Efficiently* producing *valid* test graph instances that *effectively* cover the application's branching structure requires custom techniques.

[1] GTFS is the standard open data format detailing transit schedules.

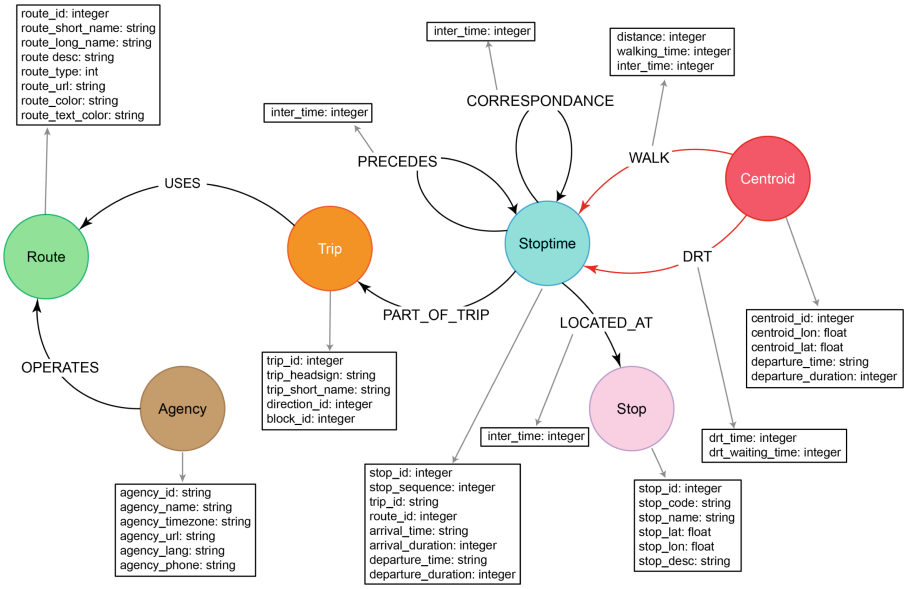

Fig. 2. Property graph schema for the P2 transport application.

Example 2 (Citation Graphs). A recent graph database use-case targets the analysis of bibliographical citation networks [3]. The schema defined by the application requires the nodes representing papers to have string values assigned to the properties that store their title, year of publication, number of referees and referrers, and PageRank. These properties are read by the application logic, and default values are substituted when they are missing, as can be seen in the code snippet in Fig. 1b. Schema-driven generators like gMark or pgMark typically generate *by design* fully schema-compliant instances where all expected properties are present, resulting in test inputs that bypass the application logic meant to handle missing or incomplete data. PGFuzz overcomes this by deliberately mutating such instances to remove or alter expected properties, triggering execution paths that would otherwise remain untested.

3 Preliminaries

Graph databases typically use the labeled property graph data model [10], i.e., a directed multi-labeled multi-graph with properties on nodes and edges.

Example 3 (Transport Graph). We illustrate the property graph model on our running example. In the P2 application, a city's transit line network is represented as a property graph. Figure 2 captures the graph database instance for the city of Royan's transport graph. The node colors correspond to their labeling. `Stoptime` nodes (blue) that represent scheduled passage times are linked

```
CREATE GRAPH TYPE transportationGraphType STRICT {
 (aType : Agency {agency_id STRING,...,agency_phone STRING}),
 (rType : Route {route_id INT32,...,route_text_color STRING}),
 (stimeType : Stoptime {stop_id INT32,..., departure_duration INT32}),
 (sType : Stop {stop_id INT32,..., stop_desc STRING}),
 (cType : Centroid {centroid_id INT32,...departure_duration INT32}), ...
 (:aType)-[:operates]->(:rType),
 (:cType)-[:DRT {drt_time INT32, drt_waiting_time INT32}]->(:stimeType),
 (:cType)
   -[:WALK {distance INT32, walking_time INT32, inter_time INT32}]->
 (:stimeType), ...
 FOR (a:Agency) EXCLUSIVE a.agency_id, FOR (s:Stop) MANDATORY s.stop_id,
 FOR (rt:Route) SINGLETON o WITHIN (a:Agency)-[o:operates]->(rt),
 FOR (c:Centroid) COUNT 2.. OF r WITHIN (s:Stoptime)-[r:DRT|WALK]->(c),
 ... }
```

Fig. 3. Simplified PG-Schema for the P2 transportation application [39].

to specific `Stop` stations (pink) through arcs labeled `LOCATED_AT`. Tessellation `Centroid` nodes (red) are connected to `Stoptime` ones via arcs labeled `WALK` and/or `DRT`, depending on whether these are accessible via pedestrian or, respectively, via demand-responsive routes. Stoptime nodes are inter-connected with arcs labeled `PRECEDES`, denoting temporal ordering, or `CORRESPONDENCE`, denoting pairwise accessibility using a different line. Each `Trip` (orange node) encodes a mobility plan that comprises passage times (`Stoptime` nodes) and employs a `Route` (green node) or several, operated by an `Agency` (brown node) or multiple ones. The property graph model also allows attaching lists of key-value properties on nodes and edges, as can be seen in Fig. 2 discussed next.

Property Graph Schemas and Constraints. While property graph instances represent *data* stored in a GDBMS, graph schemas and constraints describe their *structure*, typing, and integrity requirements to ensure consistency. Although these are crucial for data integration and exchange [19], query optimization [31], and data reliability [32], current GDBMSs provide limited support.

The applications we analyzed are built on top of Neo4j and JanusGraph. We consider these systems, as Neo4j is one of the most popular [73] GDBMSs and JanusGraph is one of the few that can handle not only a priori schema constraints but also rich cardinality ones, as surveyed in [11]. Although these systems provide different levels of support for graph schemas and constraints - Neo4j is schemaless and JanusGraph is schema flexible and allows explicit or implicit schema definitions - both benchmarks use Neo4j and JanusGraph *enforce these at the application level to maintain data integrity.*

PGFuzz supports graph schema constraints, including key constraints and cardinality ones, illustrated with the recent PG-Schema language. *Typing constraints* (PG-Types) specify nodes and edge labels and properties and the relationship types that can connect certain node labels. *Key constraints* (PG-Keys) ensure that graph objects are uniquely identified and referenced. Identification

is enforced with the `IDENTIFIER` keyword, comprising `EXCLUSIVE`, `MANDATORY`, and `SINGLETON`, while for referencing, `EXCLUSIVE MANDATORY` is used. PG-Keys statements are of the form `FOR` $p(x)$ `<qualifier>` $q(x, \bar{y})$, where `<qualifier>` is a combination of `EXCLUSIVE MANDATORY`, `SINGLETON`, and $p(x)$, $q(x, \bar{y})$ are graph queries.

We illustrate these in Fig. 3 on the running example. `EXCLUSIVE` restricts two objects from sharing a key value, e.g., no two agencies have the same `agency_id` (line 28). `MANDATORY` enforces that every object must have at least one key, e.g., every stop should have a `stop_id` (line 30). `SINGLETON` enforces that every object has at most one key, ensuring that two objects with the same key are identical, e.g., every route is operated by a unique agency (lines 32–33).

Cardinality constraints, introduced in PG-Schema as a PG-Keys extension, enforce bounds on the number of graph object instances, e.g., every centroid should be connected to a stoptime through at least two edges, labeled 'DRT' or 'WALK' (lines 35–36). The constraints are expressed similarly to PG-Keys, using the qualifier `COUNT <lower bound>?..<upper bound>? OF` to express that the number of distinct results returned by $q(x, \bar{y})$ must be within the specified range.

Among existing GDBMSs, JanusGraph provides one of the most comprehensive supports for cardinality constraints [11]. It allows declaring edge label multiplicity, specifying whether at most one (`SIMPLE`) or multiple (`MULTI`) edges can be defined between any node pair, with `MANY2ONE` and `ONE2MANY` respectively allowing at most one outgoing/incoming edge, without constraining the number of incoming/outgoing ones. It also allows for property key cardinalities, specifying whether a node key can have one or multiple values.

PGFuzz transformations allow breaking all of the above constraints, as detailed in Sect. 4.3. PGFuzz considers test inputs for graph database applications to be defined by a core fragment of the previous graph schema constraints, with typing and key constraints being enforced at the property level. As such, each property can be assigned one of the built-in types `String`, `Integer`, `Double`, `Float`, `Boolean`, with `String` being the default when no type is provided. Also, properties can be flagged as `EXCLUSIVE`, `MANDATORY`, or `SINGLETON`. Relationship cardinality constraints are also supported, enabling the specification of the allowed number of edges of a particular type between any two nodes.

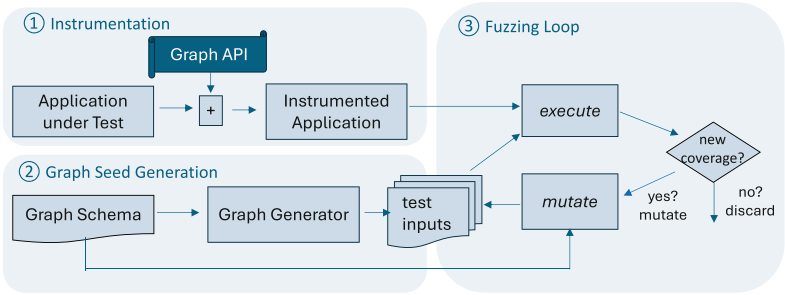

Fig. 4. The workflow of PGFuzz.

4 The PGFuzz Framework

Figure 4 illustrates the workflow of the PGFuzz testing framework. The main components of PGFuzz are: (1) program instrumentation of the application under test to direct the calls to graph databases to PGFuzz Graph API and track code coverage, (2) leveraging state-of-the-art graph generators to produce graph database input seeds, and (3) using greybox fuzzing to generate new test inputs by graph transformations guided by code coverage.

To avoid the cost of database connection, communication, and state resetting operations at each test execution, PGFuzz maintains an in-memory graph structure for the input graph database state. The model is based on a property graph model (see Sect. 3) and provides a graph processing API modeled after popular graph database query APIs. As shown in Table 1, the PGFuzz API defines the basic methods for accessing and updating graph data.

We instrument the application under test to redirect the calls to graph database operations and instead invoke the API operations on PGFuzz's internal graph. Hence, PGFuzz does not use GDBMS executions but directly explores the behaviors of the graph database application under test.

Table 1. PGFuzz API operations.

Operation	Description
`getNodes(String l, String k, String v)`	return nodes with label `l` and key-value (`k`, `v`)
`getNode(int id)`	return the node with identifier `id`
`getLabel(Node n)`	return the label attached to the node `n`
`getConnectedNodes(Node n, String l)`	return all nodes connected to `n` with edges labeled `l`
`getNodes()`, `getEdges()`	return all nodes/edges
`getNodeEdges(Node n)`	return all edges attached to the node `n`
`getIncomingEdges(Node n)`	return all incoming edges attached to the node `n`
`getOutgoingEdges(Node n)`	return all outgoing edges attached to the node `n`
`getNodePairEdges(Node n1, Node n2)`	return all edges between nodes `n1` and `n2`
`getEdgeProperty(Edge e, String l)`	return a property of an edge `e` labeled `l`
`getNodeProperty(Node n, String l)`	return a property of a node `n` labeled `l`
`getPropertyValue(String k)`	return the value of a property `k`
`addNewNode()`, `addNewNodes(int c)`	add a/`c` new node(s)
`addLabeledNode(String l)`	add a new node with label `l`
`addNode(Node n)`	add a new node `n`
`addEdge(Edge e, Node n1, Node n2)`	add the edge `e` between nodes `n1` and `n2`
`removeNode(int id)`	remove the node with identifier `id`
`removeEdge(Edge e)`	remove the edge `e`

4.1 Producing Random Graph Instances

PGFuzz constructs initial test seeds using the state-of-the-art gMark [18] and pgMark [75] schema-driven graph generators. gMark produces synthetic edge-labeled graphs based on schema constraints specifying their corresponding typing structure. The system also allows the configuration of the size of the graph, the number of occurrences for each node or edge label, and the in- and out-degree distribution for each relationship type. The pgMark framework builds on top of gMark and adds support for generating attributed graphs with node properties. In the PGFuzz framework, we use an enhanced version of pgMark, which we extended to support the *full expressiveness of the property graph model* by also allowing instances to incorporate edge properties.

4.2 Greybox Fuzzing for Graph Generation

PGFuzz collects the graph database states produced by gMark and pgMark in a set of test inputs and uses them for testing graph database applications. However, these test inputs are generated in a black-box approach, i.e., they are produced independently and randomly, not using any information from the application under test. Therefore, testing the applications using these graphs as test inputs is unlikely to cover all branching logic of the application code.

PGFuzz offers a greybox approach for generating new graph database instances for testing graph database applications. Unlike the graph instance generation by gMark and pgMark, PGFuzz employs a greybox feedback loop to guide the test generation towards more application code coverage.

As given in Fig. 4, the greybox fuzzing loop consists of three main steps: First, it runs the application under test with a test input. During the execution, it collects some greybox information (for PGFuzz, branch code coverage) of the execution. Then, it checks whether the execution covers previously unexplored branches in the application code. Following the intuition that the test inputs similar to the one that triggered new application behavior are also likely to trigger new behaviors, the fuzzer mutates that test input to generate new test inputs. PGFuzz uses graph transformations to generate new test inputs.

PGFuzz uses branch code coverage as greybox feedback information from the test executions, which is commonly used in testing traditional software systems [82]. As the application programs' branching logic is based on the input graph database state, branch code coverage effectively captures the explored parts of the application program. Alternative to code coverage, the framework can be extended to use different feedback information from the program, e.g., defining fitness functions, to evaluate how likely the test inputs generated from an input can trigger new behaviors.

4.3 Schema-Aware Graph Transformations

Given a test input, i.e., test graph database instance, that achieves new coverage PGFuzz mutates that instance to produce a new instance using novel graph

transformations that leverage the typing, key, and cardinality constraints of the application under test.

PGFuzz uses graph schema-aware transformations and supports a core fragment of the recent PG-Schema and PG-Keys formalisms, which inform the design of the novel GQL graph query standard [11,12]. PGFuzz integrates information regarding graph typing, key constraints enforced on graph objects, and cardinality restrictions, e.g., on the number of edges between node pairs, as in JanusGraph: `One2One`, `Many2One`, `One2Many`, `Many2Many`. This provides variations of graph instances that comply with application requirements and are likely to cover important states. Rather than adding a random label, the transformation can select one *that is expected to occur*. By design, the transformation can leverage information about graph constraints that might not otherwise be derived from any arbitrary graph state. Schema-aware transformations adhere to the schema when possible but may deviate if no valid candidates are available.

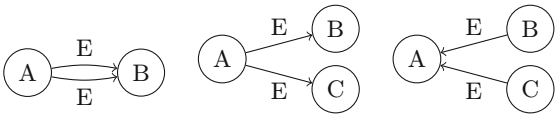

Fig. 5. $\mathbf{M_{14}}$ transformations that violate cardinality constraints. Each subfigure shows the result of a transformation breaking one of: `SIMPLE` (maximum one edge between two nodes), `MANY2ONE` (multiple sources to one target), and `ONE2MANY` (one source to multiple targets). Each graph depicts the input test state *after* the transformation, where an added edge leads to the constraint violation.

PGFuzz transformations are classified into $\mathbf{C_1}$, $\mathbf{C_2}$, $\mathbf{C_3}$, and $\mathbf{C_4}$, depending on whether they add, remove, modify graph objects, or violate graph constraints. Modifications include adding nodes or edges with arbitrary schema labels ($\mathbf{M_{1,2}}$) and assigning arbitrary schema properties ($\mathbf{M_3}$). Nodes and their attached edges can be removed ($\mathbf{M_4}$), as well as edges or properties ($\mathbf{M_{5,6}}$). Node or edge labels can be randomly modified ($\mathbf{M_{7,8}}$), along with the keys or property values ($\mathbf{M_{9,10}}$). Schema constraints can be broken by removing all nodes or edges of a given label ($\mathbf{M_{11,12}}$), altering property typing for arbitrary elements ($\mathbf{M_{13}}$), or cardinality constraints by adding or removing instances ($\mathbf{M_{14}}$ in Fig. 5), breaching exclusivity constraints by assigning values to random elements ($\mathbf{M_{15}}$), and violating mandatory constraints by setting values to `NULL` ($\mathbf{M_{16}}$).

5 Evaluation

We implement PGFuzz[2] on top of the JQF [62] fuzz testing framework for Java applications. JQF allows for instrumenting the application under test to

[2] The source code of PGFuzz and the evaluation data are available at: https://github.com/moudemans/PGFuzz.

track code coverage and implement the coverage-guided testing loop. We redirect graph database operations to PGFuzz's Graph API and then execute the instrumented program with generated graph instances as test inputs. We evaluate PGFuzz's performance compared to state-of-the-art graph instance generators, in terms of test coverage and the number of unique bugs detected, following metrics proposed in [51,54]. We test benchmark graph database-backed applications using baseline graph generators and PGFuzz, measuring branch code coverage and the number of distinct bugs discovered. We collect the test results on a machine running Windows 11 with Intel(R) Core(TM) i7-13700KF 3.40 GHz CPU and 32 GB of memory.

As baselines, we consider the state-of-the-art gMark and pgMark schema-driven graph generators for applications leveraging edge-labeled and property graphs, respectively. These tools follow a black-box testing strategy, producing test inputs independently of each other, as well as of previous test executions. PGFuzz exploits a greybox fuzzing framework to incorporate code coverage feedback into the test generation and uses schema-aware graph data transformations to generate new tests.

Table 2. Benchmark characteristics.

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

5.1 Benchmarks

We collected graph database-backed applications from *publicly available* repositories. The main criterion is ensuring that graph schema, key, and cardinality constraints are enforced in the branching logic. We curate eight benchmarks (see Table 2) from diverse areas: medical studies (P1), intelligent transportation (P2), bibliographical analysis (P3), and genomics (P5-P8), as described below.

P1 Medical is collected from OpenStudyBuilder [1], an open-source application to manage clinical data standards and study design specifications. It implements CDISC [26] standards[3] by providing additional semantics to prevent

[3] CDISC develops standards for collecting, sharing, and analyzing clinical trial data.

inconsistencies that impede automation scaling. P1 ensures that, when adding or modifying nodes, `EXCLUSIVE` and `MANDATORY` constraints are preserved.

P2 Transportation is collected from a study [2,39] that evaluates how demand-responsive services influence public transportation accessibility. P2 analyzes the connectivity of central locations on a grid with respect to mass transit stations through pedestrian and on-demand routes. Its branching logic depends on *cardinality* constraints on relationships with certain labels and properties.

P3 Citation is collected from CiteGraph [3], an open-source web application for visualizing and analyzing citation networks. It enables discovering connections between papers and authors and evaluating the influence of specific works or researchers. P3 gathers citation, collaboration, PageRank metrics, and bibliographic meta-data through graph traversals. Its logic for processing data is guided by the schema of the underlying property graph.

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

P5 Pangenomic is collected from PanGraph-DB [4], an open-source comparative pangenomics application that analyzes graph structures representing the genomic diversity of species. It inspects genomic family neighborhoods and extracts connectivity properties to identify those with similar characteristics.

P6 Pheno4J1 is collected from Pheno4J [56], an open-source application that integrates various biological data sources, allowing to visualize and analyze relationships between genetic variations and phenotypic traits. Its logic depends on the graph schema, as it computes an export format for data exchange.

P7 Pheno4J2, also collected from Pheno4J, analyses the properties linked to phenotypic traits of genomic variants. Its branching logic is determined by the edge labels of the different variants and by the properties that capture their phenotypic effects. Data integrity is maintained through type conversions.

P8 PanTool1 is collected from PanTool [5], an open-source application that analyzes pangenome graphs to evaluate genetic variations across species. It tracks and organizes phasing information from genomic sequences represented as nodes. Its branching logic depends on `MANDATORY` key constraints on node properties and cardinality constraints on the number of genomes with phasing information.

P9 PanTool2, also collected from PanTool, analyzes the messenger RNA nodes of a genomic graph. Similarly to P8, its branching logic depends on property values and `MANDATORY` key constraints on properties.

Correctness Specification and Test Oracle. We validate the benchmarks by checking whether the application crashes for certain inputs, using this property as our test oracle. We identified several bugs in the applications that led to crashes, with exceptions being thrown when specific graph database instances were used as input.

5.2 Evaluation Results

We evaluate PGFuzz by addressing the following main research questions:

- **RQ1.** Can the test inputs generated by PGFuzz provide a higher test coverage compared to the state-of-the-art graph instance generators?
- **RQ2.** Can the test inputs generated by PGFuzz detect more bugs than the state-of-the-art graph instance generators?
- **RQ3.** How do different categories of graph transformations contribute to the bug detection capability of PGFuzz?

We answer the research questions by testing the benchmark programs using the input graph instances generated by PGFuzz and the state-of-the-art graph instance generators gMark and pgMark with the synthetic graph size of 100 nodes. Due to the randomness of the tests, we conducted each experiment five times for 10 min and reported the average results.

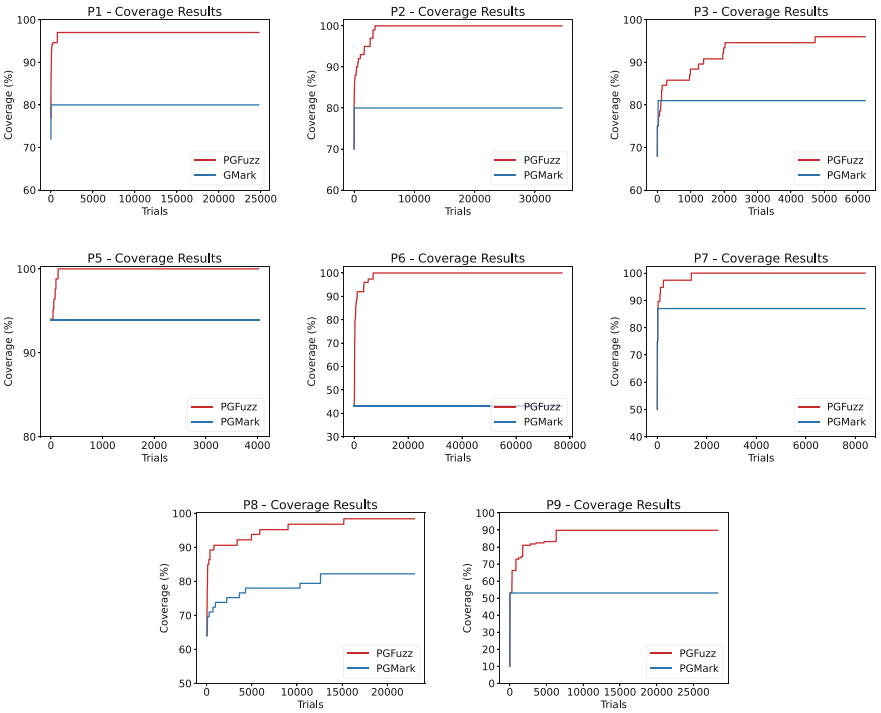

Fig. 6. Test coverage of the benchmarks using PGFuzz and the baseline.

Test Coverage. Figure 6 shows the branch code coverage of testing the benchmark applications using gMark (for P1, which uses an edge-labeled graph), pgMark (for P2-P9, which use property graphs), and PGFuzz. The plots do not include the coverage results for benchmark P4 since all branches in that benchmark program are covered within the first few test cases generated by

each method. The results show that testing the benchmarks with the black-box testing approach using pgMark reaches 40%–85% branch coverage. The broad coverage of the application logic achieved with randomly generated instances can be attributed to the diversity of graph instances gMark and pgMark produce by design, as demonstrated by their empirical evaluation [18]. However, the generated graph instances do not cover much of the code in all benchmarks, e.g., in P6 and P9. These benchmarks use various instance nodes, edges, relation labels, and properties in their application logic and use nested branching on these properties. Increasing the test coverage of such benchmarks requires exploring a more extensive set of graph instances. Coverage-guided fuzzing approaches increase the code coverage by generating more graph instances around the instances that hit new code branches, hence exploring deeper branches of the applications.

We answer **RQ1** positively: PGFuzz achieves better coverage results in a smaller number of test executions for all the benchmark applications. PGFuzz's schema-aware test case generation produces test instances that comply with schema types but intentionally break the schema constraints. An example is the branching given in Sect. 3 and Fig. 1b, which requires the exploration of graph instances that do not necessarily satisfy all the constraints in the graph schema. The graph transformation-based mutations of PGFuzz lead to covering the application logic with the substituted values for the graph properties. Moreover, transforming the instances that cover new branches helps in covering a deeper sequence of branching. PGFuzz shows an increase in coverage of up to 57% and 23% on average, with the highest improvement in benchmarks P6 and P8 that have the most complicated branching logic.

While PGFuzz consistently outperforms the baselines, it does not reach 100% coverage for all benchmarks. Further analysis of the applications shows that P1 has an unreachable condition in a private method, as it has already been checked and filtered in an earlier branch. In P3, PGFuzz can reach 100% coverage but cannot consistently cover each branch in all test repetitions. Branches that are not consistently reached depend on the presence of a specific node and require input graph transformations that enable access but make reaching them unlikely. In P9, PGFuzz cannot reach 100% coverage, as doing so would require sequential node property IDs, which are highly improbable to occur in both randomly generated and transformed inputs.

Bug Detection. We answer **RQ2** positively: PGFuzz detects more bugs than testing the applications using the state-of-the-art graph instance generators. In our evaluation, PGFuzz found several bugs in the benchmark applications, while the baseline approaches failed to detect any of them.

Table 3 lists the number of distinct bugs PGFuzz discovered. The most frequent errors, encountered 23 times, are due to invalid data types, as the benchmark tries to parse integers, booleans, or other types while the property value is no longer of that type. Null pointer exceptions occurred 5 times and were triggered by certain graph elements no longer being present. The array index out of bounds occurred 3 times and was caused by removing graph elements or splitting a string that no longer contained a specific character. These errors are

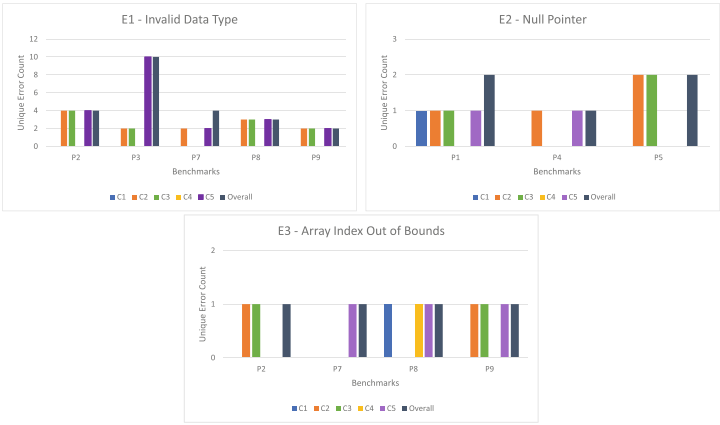

Fig. 7. Contribution of the transformation categories per error type.

thrown by the application under test when the contents of the graph instance trigger parts of the application code that do not anticipate certain nodes, edges, labels, or properties. Although the errors seem simple, they are hard to detect as they manifest only with certain contents of the graph instances to hit certain branches that are neglected by the developers.

Overall, PGFuzz is successful at finding bugs, especially with the graph transformations that break schema constraints. PGFuzz could expose several errors that are caused mainly by changing the graph elements and property types, producing possible test inputs that are neglected in the application logic.

Table 3. Distinct bugs per method causing Null Pointer Exception (NPE), Number Format Exception (NFE), Array Index Out of Bounds Exception (AIOBE).

	PGFuzz	Error(s)
P1	2	NPE
P2	5	AIOBE, NFE
P3	10	NFE
P4	1	NPE
P5	2	NPE
P6	0	-
P7	4	AIOBE, NFE
P8	4	AIOBE, NFE
P9	3	AIOBE, NFE

Contribution of the Transformation Categories for Bug Detection. To answer **RQ3**, we analyzed the effect of the categories of transformations to detect each type of error. Figure 7 shows the number of errors hit by each category of transformations for (1) an invalid data type, (2) a null pointer, and (3) an array index out-of-bounds exception. Breaking graph constraints (C4), which potentially violate the schema constraints of an application, is the most effective strategy, detecting the most unique errors in 6 of 8 benchmarks. This underscores PGFuzz's schema-aware graph transformations, which generate schema-compliant yet semantically unexpected graph instances that violate constraints.

Compound Transformations. The graph transformations in PGFuzz focus on modifying individual graph elements, such as nodes or edges, along with their corresponding labels, properties, or values, as described in Sect. 4.3. To evaluate the effectiveness of making multiple changes to the graph instances, we introduced *compound transformations*. We then repeated the tests by incorporating these into PGFuzz's set of transformations.

Note that compound transformations do not change individual components of a graph; instead, they apply multiple modifications at once. Specifically, we used compound transformations that involve randomly adding multiple nodes and edges in the input graph. This process results in the addition of several new nodes, edges, and relationships that are likely to form more complex structures such as cycles or triangles. We avoid using transformations that remove subsets of graph elements, as such operations can be overly disruptive.

We repeated the evaluation on the set of benchmarks, enabling compound transformations. As given in Fig. 8, the results do not show a significant performance improvement on the application benchmarks. This can be explained by the branching logic in graph database applications, which can be covered by modifying a single element of the test input graph, e.g., labels or properties. Compound mutations can be more beneficial for the applications that branch on more specific patterns, e.g., checking for cycles or triangles.

Threats to Validity. Potential threats to the validity of our findings include the representativeness of the curated set of database-backed application benchmarks and the randomness of the test executions. The generalizability of our results highly depends on how representative our evaluation targets are of real-world application scenarios. As we collect the benchmark suite from open-source repositories, we do not cover enterprise applications. However, our benchmark applications have diverse graph database schemas and schema constraints.

An additional threat to the validity may arise from the decision to redirect database calls to PGFuzz API operations. The effectiveness of this approach depends on the assumption that the API calls accurately represent the behavior of the database methods. Any discrepancies between the two could negatively impact PGFuzz's performance. The API operations model all the database calls in the benchmark programs used in our experiments.

Lastly, the experimental results involve multiple sources of randomness. The input graph instances are randomly generated by gMark and pgMark, which

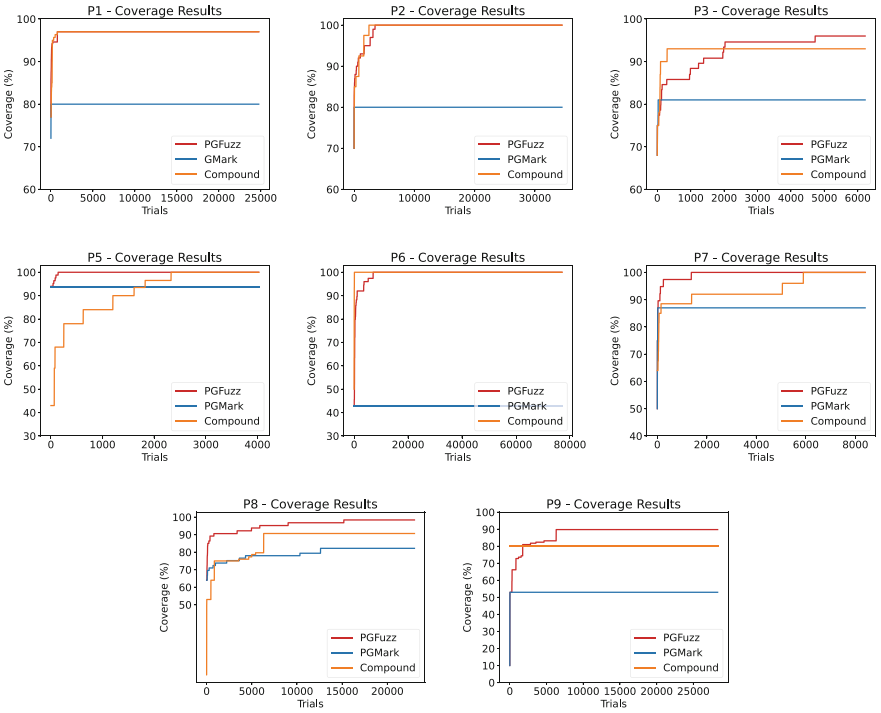

Fig. 8. Test coverage of the benchmarks results using compound transformations.

are later transformed by PGFuzz. Moreover, the effectiveness of PGFuzz's tests is highly dependent on the applied transformations, which are also selected randomly as part of the fuzzing framework. To mitigate randomness, we repeat our experiments five times and report averaged results.

6 Related Work

Testing Database Systems. Several recent works focus on testing database management systems [9,17,45] and graph databases [47,49,83]. Different from these works, which target the bugs in the database systems, PGFuzz targets finding bugs in the graph database-backed applications.

Testing Database-Backed Applications. As the behavior of database-backed applications depends on the state of the database on which they operate, testing them requires generating test database instances. Earlier works [6,30] use constraint solving and static analysis for generating program inputs and input database states. SynDB [63,64] tests database-backed applications using a symbolic database. It generates test inputs and relational database states by considering all program, query, and database constraints and targeting relational databases. Recent work proposes DBGRILLER [81], a greybox testing approach for generating test databases for database-backed applications. Similar to

PGFuzz, DBGRILLER applies small mutations to existing database states to create new test instances. However, it is specifically designed for relational database applications and does not tackle graph-database-backed applications. To our knowledge, PGFuzz is the first greybox fuzzer for generating test inputs for graph database-backed applications.

Fuzz Testing. There is extensive work on generating test input using fuzzing [52,82,84]. While the first fuzzers, such as American Fuzzy Lop (AFL) [79] and libFuzzer [80], work on the bit-level representation of the input seed files, they are not as effective on structured inputs, such as XML and JavaScript, as bit-level mutations either disrupt the structure or are unlikely to produce changes beneficial to code coverage. Smart fuzzing [66] and grammar-based fuzzing [78] operate on test input structures and produce test inputs for a given input structure. These approaches have been successfully used to generate test inputs in various formats [34,48,59,74,78]. GraphFuzz [37] addresses library API testing by modeling executed functions as a dataflow graph and applying graph-based mutations to generate new test cases. Although GraphFuzz also uses graph mutations to modify test cases, it is tailored to dataflows and fundamentally differs from PGFuzz. Similar to smart and grammar-based fuzzers, PGFuzz generates syntactically valid test inputs, satisfying the syntactical schema type of the underlying graph database. Unlike grammar-based fuzzers that aim to cover the space of the *syntactic* structure, enforcing constraints in the generated input, PGFuzz can generate test inputs that may intentionally break *semantic* constraints that capture the intended logic of the data, such as key and cardinality constraints. This enables targeted testing of graph database applications where such constraints play a critical role in ensuring data integrity.

Graph Databases. Graph transformations have been extensively applied to schema evolution, data migration, and interoperability in graph databases. Bonifati et al. [23] propose a framework for supporting evolving graph schemas, which leverages graph rewriting operations to maintain consistency. More recent works [24,25] adopt a high-level, declarative approach aligned with the GQL standard, employing graph pattern matching to support complex migration and cleaning tasks. Similarly, schema-aware transformations ensure semantic interoperability between diverse graph models such as property graphs and RDF [68]. In contrast, PGFuzz leverages a lightweight, *heuristic* use of schema-aware transformations aimed at generating varied test instances for evaluating the robustness of graph database-backed applications.

7 Conclusion

The rise of graph database management systems emphasizes the importance of testing graph database-backed applications that rely on complex, highly interconnected datasets. Our work introduces PGFuzz, the first greybox fuzz-testing framework designed to specifically address the intricacies of testing graph database-backed applications. By incorporating schema-aware transformations

and leveraging code coverage, PGFuzz offers a promising solution to testing graph database applications. Our empirical evaluations demonstrate that PGFuzz significantly outperforms existing state-of-the-art baselines, uncovering more distinct bugs and achieving higher test coverage in graph database applications from various domains. PGFuzz opens the promising perspective of also incorporating other recently introduced types of graph constraints [31].

Acknowledgments. This work was partially supported by the grant ANR-24-CE25-1109 (Dumbrava).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. OpenStudyBuilder (2022). https://gitlab.com/Novo-Nordisk/nn-public/openstudybuilder/project-description
2. Transport Network Graph Database (2022). https://github.com/CathiaLH/GraphDatabaseCombinedTransportNetwork
3. CiteGraph (2023). https://github.com/citegraph/citegraph
4. PanGraph-DB (2023). https://github.com/jpjarnoux/PanGraph-DB
5. PanTool (2023). https://git.wur.nl/bioinformatics/pantools/
6. Agrawal, P., Chandra, B., Emani, K.V., Garg, N., Sudarshan, S.: Test data generation for database applications. In: ICDE, pp. 1621–1624. IEEE Computer Society (2018)
7. Al-Saleem, J., et al.: Knowledge graph-based approaches to drug repurposing for COVID-19. J. Chem. Inf. Model. **61**(8), 4058–4067 (2021)
8. AllegroGraph: AllegroGraph. https://allegrograph.com/. Visited 2024
9. Alvaro, P., Rigger, M.: Automatically testing database systems: DBMS testing with test oracles, transaction history, and fuzzing. ACM Queue **21**(6), 128–135 (2024)
10. Angles, R.: The property graph database model. In: AMW. CEUR Workshop Proceedings, vol. 2100. CEUR-WS.org (2018)
11. Angles, R., et al.: PG-schema: schemas for property graphs. Proc. ACM Manag. Data **1**(2), 198:1–198:25 (2023)
12. Angles, R., et al.: PG-keys: keys for property graphs. In: SIGMOD Conference, pp. 2423–2436. ACM (2021)
13. ArangoDB: ArangoDB. https://arangodb.com/. Visited 2024
14. Arnoux, J., Bonifati, A., Calteau, A., Dumbrava, S., Gautreau, G.: Integrating complex pangenome graphs. In: ICDEW, pp. 350–354. IEEE (2024)
15. Atzeni, P., Bellomarini, L., Iezzi, M., Sallinger, E., Vlad, A.: Weaving enterprise knowledge graphs: the case of company ownership graphs. In: EDBT, pp. 555–566. OpenProceedings.org (2020)
16. AWS: Amazon Neptune. https://aws.amazon.com/fr/neptune/. Visited 2024
17. Ba, J., Rigger, M.: Testing database engines via query plan guidance. In: ICSE, pp. 2060–2071. IEEE (2023)
18. Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G.H., Lemay, A., Advokaat, N.: gMark: schema-driven generation of graphs and queries. IEEE Trans. Knowl. Data Eng. **29**(4), 856–869 (2016)

19. Barceló, P., Pérez, J., Reutter, J.L.: Schema mappings and data exchange for graph databases. In: ICDT, pp. 189–200. ACM (2013)
20. Bitnine. Co., Ltd.: AgensGraph. https://bitnine.net/agensgraph. Visited 2024
21. BlazeGraph: BlazeGraph. https://blazegraph.com/. Visited 2024
22. Bohannon, P., Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Conditional functional dependencies for data cleaning. In: ICDE, pp. 746–755. IEEE Computer Society (2007)
23. Bonifati, A., Furniss, P., Green, A., Harmer, R., Oshurko, E., Voigt, H.: Schema validation and evolution for graph databases. In: Laender, A., Pernici, B., Lim, E.-P., de Oliveira, J. (eds.) ER 2019. LNCS, vol. 11788, pp. 448–456. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-33223-5_37
24. Bonifati, A., Murlak, F., Ramusat, Y.: Transforming property graphs. Proc. VLDB Endow. **17**(11), 2906–2918 (2024)
25. Bonifati, A., Ramusat, Y., Murlak, F., Fejza, A., Echahed, R.: DTGraph: declarative transformations of property graphs. Proc. VLDB Endow. **17**(12), 4265–4268 (2024)
26. Clinical Data Interchange Standards Consortium: CDISC (2022). https://www.cdisc.org/. Visited 2024
27. DataStax: DataStax Enterprise Graph. https://www.datastax.com/products/datastax-graph. Visited 2024
28. DGraph: DGraph. https://dgraph.io/. Visited 2024
29. Elayam, M.M., Ray, C., Claramunt, C.: A hierarchical graph-based model for mobility data representation and analysis. Data Knowl. Eng. **141**, 102054 (2022)
30. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: ISSTA, pp. 151–162. ACM (2007)
31. Fan, W.: Dependencies for graphs: challenges and opportunities. ACM J. Data Inf. Qual. **11**(2), 5:1–5:12 (2019)
32. Fan, W., Geerts, F.: Foundations of Data Quality Management. Synthesis Lectures on Data Management. Morgan & Claypool Publishers (2012)
33. Francis, N., Libkin, L.: Schema mappings for data graphs. In: PODS, pp. 389–401. ACM (2017)
34. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based Whitebox fuzzing. In: PLDI, pp. 206–215. ACM (2008)
35. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated Whitebox fuzz testing. In: NDSS. The Internet Society (2008)
36. Gosnell, D., Broecheler, M.: The practitioner's guide to graph data. https://www.oreilly.com/library/view/the-practitioners-guide/9781492044062/. Visited 2024
37. Green, H., Avgerinos, T.: GraphFuzz: library API fuzzing with lifetime-aware dataflow graphs, vol. 2022, pp. 1070–1081. IEEE Computer Society (2022). https://doi.org/10.1145/3510003.3510228
38. Gütebier, L., et al.: CovidGraph: a graph to fight COVID-19. Bioinform. **38**(20), 4843–4845 (2022)
39. Hasif, C.L., Araldo, A., Dumbrava, S., Watel, D.: A graph-database approach to assess the impact of demand-responsive services on public transit accessibility. In: IWCTS@SIGSPATIAL, pp. 2:1–2:4. ACM (2022)
40. Hegeman, T., Iosup, A.: Survey of graph analysis applications. CoRR abs/1807.00382 (2018)
41. Huang, H., Bucher, D., Kissling, J., Weibel, R., Raubal, M.: Multimodal route planning with public transport and carpooling. IEEE Trans. Intell. Transp. Syst. **20**(9), 3513–3525 (2019)

42. HyperGraphDB: HyperGraphDB. https://hypergraphdb.org/. Visited 2024
43. IBM: DB2 Graph. https://www.ibm.com/docs/en/db2-warehouse?topic=applications-db2-graph. Visited 2024
44. Ilyas, I.F., Chu, X.: Data Cleaning. ACM Books, vol. 28. ACM (2019)
45. Ba, J., Rigger, M.: Keep it simple: testing databases via differential query plans. Proc. ACM Manag. Data **2**(3), 188 (2024)
46. JanusGraph: JanusGraph. https://janusgraph.org/. Visited 2024
47. Jiang, Y., Liu, J., Ba, J., Yap, R.H.C., Liang, Z., Rigger, M.: Detecting logic bugs in graph database management systems via injective and surjective graph query transformation. In: ICSE, pp. 46:1–46:12. ACM (2024)
48. Borges Jr., N.P., Havrikov, N., Zeller, A.: Generating tests that cover input structure. In: Software Engineering. LNI, vol. P-310, pp. 85–86. Gesellschaft für Informatik e.V. (2021)
49. Kamm, M., Rigger, M., Zhang, C., Su, Z.: Testing graph database engines via query partitioning. In: ISSTA, pp. 140–149. ACM (2023)
50. Kertkeidkachorn, N., Nararatwong, R., Xu, Z., Ichise, R.: FinKG: a core financial knowledge graph for financial analysis. In: ICSC, pp. 90–93. IEEE (2023)
51. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: CCS, pp. 2123–2138. ACM (2018)
52. Li, J., Zhao, B., Zhang, C.: Fuzzing: a survey. Cybersecurity **1**(1), 1–13 (2018). https://doi.org/10.1186/s42400-018-0002-y
53. MemGraph: MemGraph. https://memgraph.com/. Visited 2024
54. Metzman, J., Szekeres, L., Simon, L., Sprabery, R., Arya, A.: FuzzBench: an open fuzzer benchmarking platform and service. In: ESEC/SIGSOFT FSE, pp. 1393–1403. ACM (2021)
55. Microsoft: Azure Cosmos DB. https://azure.microsoft.com/fr-fr/products/cosmos-db. Visited 2024
56. Mughal, S., Moghul, I., Yu, J., Clark, T., Gregory, D.S., Pontikos, N.: Pheno4J: a gene to phenotype graph database. Bioinformatics **33**(20), 3317–3319 (2017)
57. Neo4J: Neo4J. https://neo4j.com/. Visited 2024
58. Noy, N.F., Gao, Y., Jain, A., Narayanan, A., Patterson, A., Taylor, J.: Industry-scale knowledge graphs: lessons and challenges. Commun. ACM **62**(8), 36–43 (2019)
59. Olsthoorn, M., van Deursen, A., Panichella, A.: Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In: ASE, pp. 1224–1228. IEEE (2020)
60. Oracle: Oracle Big Data Spatial and Graph. https://www.oracle.com/database/technologies/bigdata-spatialandgraph.html. Visited 2024
61. OrientDB: OrientDB. http://orientdb.org/. Visited 2024
62. Padhye, R., Lemieux, C., Sen, K., Papadakis, M., Traon, Y.L.: Semantic fuzzing with zest. In: ISSTA, pp. 329–340. ACM (2019)
63. Pan, K., Wu, X., Xie, T.: Automatic test generation for mutation testing on database applications. In: AST, pp. 111–117. IEEE Computer Society (2013)
64. Pan, K., Wu, X., Xie, T.: Guided test generation for database applications via synthesized database interactions. ACM Trans. Softw. Eng. Methodol. **23**(2), 12:1–12:27 (2014)
65. Park, S., Cheng, T.: Framework for constructing multimodal transport networks and routing using a graph database: a case study in London. Trans. GIS **27**(5), 1391–1417 (2023)
66. Pham, V., Böhme, M., Santosa, A.E., Caciulescu, A.R., Roychoudhury, A.: Smart greybox fuzzing. IEEE Trans. Software Eng. **47**(9), 1980–1997 (2021)

67. Preusse, M., et al.: COVIDGraph: connecting biomedical COVID-19 resources and computational biology models. In: SEA-Data@VLDB. CEUR Workshop Proceedings, vol. 2929, pp. 34–37. CEUR-WS.org (2021)
68. Rabbani, K., Lissandrini, M., Bonifati, A., Hose, K.: Transforming RDF graphs to property graphs using standardized schemas. Proc. ACM Manag. Data **2**(6), 242:1–242:25 (2024)
69. RedisGraph: RedisGraph. https://redis.io/. Visited 2024
70. Sahu, S., Mhedhbi, A., Salihoglu, S., Lin, J., Özsu, M.T.: The ubiquity of large graphs and surprising challenges of graph processing: extended survey. VLDB J. **29**(2–3), 595–618 (2020)
71. Sakr, S., et al.: The future is big graphs: a community view on graph processing systems. Commun. ACM **64**(9), 62–71 (2021)
72. Sequeda, J., Lassila, O.: Designing and building enterprise knowledge graphs. In: Synthesis Lectures on Data Semantics, and Knowledge. Morgan & Claypool Publishers (2021)
73. Redgate Software: Neo4J (2024). https://db-engines.com/en/ranking/graph+dbms. Visited 2024
74. Steinhöfel, D., Zeller, A.: Input invariants. In: ESEC/SIGSOFT FSE, pp. 583–594. ACM (2022)
75. Thom Hurks: PGMark: a domain-independent tool for generating property graphs based on a user-defined schema. https://github.com/ThomHurks/pgMark. Visited 2024
76. TigerGraph: TigerGraph. https://www.tigergraph.com/. Visited 2024
77. Titan: Titan. http://espeed.github.io/titandb/. Visited 2024
78. Wang, J., Chen, B., Wei, L., Liu, Y.: Superion: grammar-aware greybox fuzzing. In: ICSE, pp. 724–735. IEEE/ACM (2019)
79. Website, A.: American Fuzzy Loop. http://lcamtuf.coredump.cx/afl/. Accessed 2024
80. AFL Website: libFuzzer: A library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html. Accessed 2024
81. Yan, C., Nath, S., Lu, S.: Generating test databases for database-backed applications. In: ICSE, pp. 2048–2059. IEEE (2023)
82. Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C.: The fuzzing book (2019)
83. Zheng, Y., et al.: Differential optimization testing of Gremlin-based graph database systems. In: ICST, pp. 25–36. IEEE (2024)
84. Zhu, X., Wen, S., Camtepe, S., Xiang, Y.: Fuzzing: a survey for roadmap. ACM Comput. Surv. (CSUR) **54**(11s), 1–36 (2022)