Finding Bugs in Gremlin-Based Graph Database Systems via Randomized Differential Testing

Yingying Zheng
State Key Lab of Computer Science at ISCAS, University of CAS, China
zhengyingying14@otcaix.iscas.ac.cn

Wensheng Dou*
State Key Lab of Computer Science at ISCAS, University of CAS, University of CAS Nanjing College, China
wsdou@otcaix.iscas.ac.cn

Yicheng Wang
State Key Lab of Computer Science at ISCAS, University of CAS, China
wangyicheng19@otcaix.iscas.ac.cn

Zheng Qin
State Key Lab of Computer Science at ISCAS, University of CAS, China
qinzheng19@otcaix.iscas.ac.cn

Lei Tang
State Key Lab of Computer Science at ISCAS, University of CAS, China
tanglei20@otcaix.iscas.ac.cn

Yu Gao
State Key Lab of Computer Science at ISCAS, University of CAS, China
gaoyu15@otcaix.iscas.ac.cn

Dong Wang
State Key Lab of Computer Science at ISCAS, University of CAS, China
wangdong18@otcaix.iscas.ac.cn

Wei Wang*
State Key Lab of Computer Science at ISCAS, University of CAS, University of CAS Nanjing College, China
wangwei@otcaix.iscas.ac.cn

Jun Wei
State Key Lab of Computer Science at ISCAS, University of CAS, Nanjing Institute of Software Technology, China
wj@otcaix.iscas.ac.cn

ABSTRACT

Graph database systems (GDBs) allow efficiently storing and retrieving graph data, and have become the critical component in many applications, e.g., knowledge graphs, social networks, and fraud detection. It is important to ensure that GDBs operate correctly. Logic bugs can occur and make GDBs return an incorrect result for a given query. These bugs are critical and can easily go unnoticed by developers when the graph and queries become complicated. Despite the importance of GDBs, logic bugs in GDBs have received less attention than those in relational database systems.

In this paper, we present *Grand*, an approach for automatically finding logic bugs in GDBs that adopt Gremlin as their query language. The core idea of Grand is to construct semantically equivalent databases for multiple GDBs, and then compare the results of a Gremlin query on these databases. If the return results of a query on multiple GDBs are different, the likely cause is a logic bug in these GDBs. To effectively test GDBs, we propose a model-based query generation approach to generate valid Gremlin queries that can potentially return non-empty results, and a data mapping approach to unify the format of query results for different GDBs. We evaluate Grand on six widely-used GDBs, e.g., Neo4j and HugeGraph. In total, we have found 21 previously-unknown logic bugs in these GDBs. Among them, developers have confirmed 18 bugs, and fixed 7 bugs.

CCS CONCEPTS

• **Information systems** → **Database query processing**; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Graph database systems, differential testing, Gremlin

ACM Reference Format:
Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. 2022. 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 (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534409

1 INTRODUCTION

Graph database systems (GDBs) are built on the labeled property graph model [55] or the Resource Description Framework (RDF) graph model [25], and support efficient storage and queries for graph data, which consists of vertices and edges. The popularity of GDBs has increased dramatically recently, and GDBs have played a significant role in many applications, e.g., knowledge graphs [28, 43], social networks [33], and fraud detection [47]. Examples of the most popular GDBs include Neo4j [7], OrientDB [11], JanusGraph [6] (extended from Titan [20]), Nebula [10], HugeGraph [3], TinkerGraph [16], ArcadeDB [9] and so on.

Similar to relational database systems, GDBs also suffer from logic bugs, in which a query returns an unexpected result without crashing the GDBs. The unexpected results could be incorrect query results (e.g., omitting a vertex in a graph), or unexpected

*Wensheng Dou and Wei Wang are the corresponding authors. CAS is the abbreviation of Chinese Academy of Sciences. ISCAS is the abbreviation of Institute of Software, Chinese Academy of Sciences.

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534409

```
1  hugegraph.schema().vertexLabel("vLabel").properties("prop")
       .nullableKeys("prop").create();
2  hugegraph.schema().indexLabel("index").onV("vLabel").by("
       prop").shard().ifNotExist().create();
3
4  Vertex v1 = new Vertex("vLabel").property("prop", 5);
5  Vertex v2 = new Vertex("vLabel").property("prop", 1);
6  Vertex v3 = new Vertex("vLabel").property("prop", 3);
7  addVertices(Arrays.asList(v1, v2, v3));
8
9  g.V().has('prop', between(0,4).or(lt(2))).count();
10    -- {3} ✘     {2} ✔
```

Figure 1: An illustrative Gremlin query that triggers a logic bug in HugeGraph.

errors. Figure 1 shows a test case that triggers a logic bug, which we found in HugeGraph [3]. In this test case, we first create a graph schema with a vertex label *vLabel* and its property *prop* (Line 1). We further create an *index* on property *prop* (Line 2). Based on this graph schema, we create three vertices $v1$, $v2$ and $v3$ with different *prop* of 5, 1, and 3, respectively (Line 4-6), and add them into the graph (Line 7). We count the vertices whose property *prop* is between 0 and 4, or less than 2 (Line 9). We can see that vertex $v2$ and $v3$ satisfy this condition, and this query should return 2. However, HugeGraph returns an incorrect result 3 because HugeGraph forgets to deduplicate overlapping values for OR operation.

For relational database systems that utilize Structured Query Language (SQL) to create, access, and modify data, researchers have developed several tools, such as RAGS [52] and SQLancer [13], to effectively discover logic bugs. RAGS uses differential testing for detecting bugs in relational database systems, while SQLancer offers three oracles, i.e., Pivoted Query Synthesis (PQS) [50], Ternary Logic Partitioning (TLP) [49], and Non-Optimizing Reference Engine Construction (NoREC) [48] to find logic bugs. However, no available tools can be applied on GDBs to detect logic bugs.

Logic bugs in GDBs are difficult to detect automatically due to the following reasons: (1) Unlike relational database systems, there is no standardized way to query graph data. These widely-used GDBs usually utilize their own query languages. For example, Neo4j develops Cypher [36], and TigerGraph uses GSQL [32]. Fortunately, most GDBs, e.g., 66% (23/35) GDBs in the DB-Engine Ranking of Graph DBMSs [22], support a common query language, Gremlin [51], which is developed by Apache TinkerPop [17] and provides a group of Gremlin APIs to query graph data. (2) GDBs adopt totally different syntax and query patterns from SQL. For example, Gremlin is a functional language and traverses graphs through a sequence of traversal steps that can be composed to express complex queries. Thus, we cannot directly apply existing testing approaches for relational database systems on GDBs. (3) GDBs have different storage and query result formats. For example, different ID generation strategies in GDBs can lead to different query results for vertices and edges. Thus, a key challenge is to automatically construct valid graph queries and their corresponding oracles for different GDBs.

In this paper, we propose *Grand*, a randomized differential testing approach for automatically finding logic bugs in GDBs that adopt Gremlin to retrieve graph data (also called Gremlin-based GDBs). Specially, by generating random Gremlin queries, our approach seeks for discrepancies among the return results from different Gremlin-based GDBs, and then identifies discrepancies as potential logic bugs. To effectively find logic bugs in Gremlin-based GDBs, we address two specific challenges. First, to generate syntactically correct and valid Gremlin queries that can return non-empty query results with a high possibility, we adopt a model-based query generation approach. Specifically, we summarize Gremlin traversal APIs to different traversal types. Then, we propose a Gremlin traversal model that expresses the transformation between different traversal APIs to generate correct and valid Gremlin queries. Second, to obtain the uniform query results from different GDBs, we utilize a data mapping approach to obtain uniform query results in different GDBs. Thus, our approach can compare query results with different formats returned by different GDBs.

To the best of our knowledge, Grand is the first approach to detect logic bugs in GDBs. To evaluate the effectiveness of Grand, we apply it on six widely-used GDBs, i.e., Neo4j [7], OrientDB [11], JanusGraph [6], HugeGraph [3], TinkerGraph [16], and ArcadeDB [9]. The experimental results show that Grand is effective in detecting logic bugs in GDBs. At the time of writing this paper, Grand has detected 21 previously-unknown logic bugs in these GDBs. Among these bugs, 18 bugs have already been confirmed, and 7 bugs have been fixed. We have made Grand publicly available at https://github.com/tcse-iscas/Grand.

In summary, this paper makes the following contributions.

- We propose Grand, an automated differential testing approach to find logic bugs in Gremlin-based graph database systems. We introduce a model-based query generation approach to effectively generate syntactically correct and valid Gremlin queries.
- We implement Grand and evaluate it on six widely-used GDBs and discover 21 previously-unknown bugs.

The remainder of this paper is organized as follows. Section 2 briefly introduces graph models and graph query languages in graph database systems. Section 3 presents the techniques of our approach including the graph database generation, the model-based query generation and differential testing for Gremlin-based GDBs. Section 4 introduces our implementation. The evaluation of Grand is shown in Section 5, followed by the threats and limitations of our approach in Section 6. Section 7 presents related work and Section 8 concludes this paper.

2 PRELIMINARIES

In this section, we introduce graph models and query languages used in graph database systems.

2.1 Graph Models in Graph Database Systems

Graph database systems (GDBs) adopt graph data structure to store and query data [55]. To explicitly lay out the associated relationships between vertices of data, GDBs use several graph models, which mainly include labeled property graph model and Resource Description Framework (RDF) graph model.

Labeled property graph model. The labeled property graph describes the relationship between entities, and consists of a set of *vertices*, a set of *edges* (i.e., the relationships of these vertices),

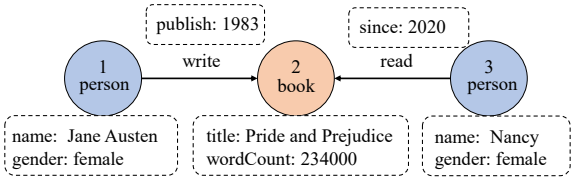

Figure 2: A labeled property graph.

labels (i.e., the groups of vertices or edges), and *properties* (i.e., attributes). Specially, each vertex or edge has a set of properties associated with it, and can be divided into different groups by its label. For example, as shown in Figure 2, two vertices with label *person* (i.e., *vertex*1 and *vertex*3) have *name* and *gender* properties, while a vertex with label *book* has *title* and *wordCount* properties. Two edges labeled by *write* and *read*, associate *person* and *book* vertices, and have *publish* and *since* properties, respectively. Edges in a labeled property graph are directed. For example, the edge labeled by *write* points from *vertex*1 to *vertex*2, while the edge labeled by *read* points from *vertex*3 to *vertex*2.

RDF graph model. The RDF graph model is a W3C standard for data exchange and uses triples (i.e., subject, object and predicate) to represent resources in the Web. Different from labeled property graph model, vertices and edges play absolutely the same role in RDF [12]. In RDF triple structure, *subject* can be a vertex or an edge, *object* is another vertex, edge or a literal value, and *predicate* describes the relationship between them. For example, a triple (*Nancy person name*) means that Nancy is a person name.

These two graph models provide different ways to describe connected data, and can be used in different applications with their own strength. For example, in knowledge graph, RDF is widely used for its highly structured information. However, in graph database systems, the labeled property graph model is more popular [22]. Specifically, almost all GDBs support the labeled property graph model, e.g., Neo4j [7], JanusGraph [6], and TigerGraph [1], while less than half of GDBs, e.g., Virtuoso [34], support the RDF graph model. Thus, we focus on the labeled property graph model in our research.

2.2 Graph Query Languages

Different from relational database systems, which use SQL as a universal way to create, access, and modify data, there is no standardized way in GDBs to query a labeled property graph. To efficiently and conveniently access the data stored in the labeled property graph, some graph query languages are developed [40]. According to their design principles, we can divide these query languages into two categories.

Category 1: SQL-like graph query language. Different SQL-like graph query languages are developed for different GDBs, which focus on individual GDBs' requirements. (1) The Cypher graph query language [36] is designed for Neo4j [2] and has been extracted as an open source project openCypher [21]. It uses *MATCH* (a pattern matching clause), *WHERE* (a filter clause) and *RETURN* (a transformation clause) as its query syntax. For example, we can use the query in Figure 3 to access the writer whose book is read by Nancy in Figure 2. (2) Nebula [10] uses nGQL which uses *FETCH*

```
1 MATCH (Person {name:'Nancy'})-[read]->(book)<-[write]-(p:
      Person)
2 RETURN p.name
```

Figure 3: An example of Cypher graph query language.

```
1 g.V().where(values('name').is(eq('Nancy')))
2   .outE('read').inV()
3   .inE('write').outV().values('name')
```

Figure 4: An example of Gremlin query language.

to query the labeled property graph. (3) TigerGraph [15] proposes GSQL [32], which uses *SELECT*, *FROM* and *WHERE* to obtain data. A GSQL query usually contains one or more SELECT statements. Most SQL-like declarative query languages are implemented for specific GDBs, and cannot be supported by other GDBs.

Category 2: Functional graph query language. Gremlin [51] is a functional, data-flow query language for traversing labeled property graphs and enables users to express complex traversals by composing a sequence of Gremlin steps (i.e., Gremlin API calls). Gremlin is introduced by Apache TinkerPop framework [54], and has been widely used in most popular GDBs, e.g., Neo4j [7], JanusGraph [6], TinkerGraph [16], and HugeGraph [3].

We can use the query in Figure 4 to get the same result as the Cypher query in Figure 3. Specifically, we can first get all vertices using *g.V()*. Then we filter the person whose name is `Nancy` using *where()* API. The book that Nancy has read can be accessed by *outE('read').inV()*. Finally, the writer can be obtained by *inE('write').outV().values('name')*. In this case, a nested query exists in *where()* API, in which *is()* is used to judge whether a property value matches the predicate, i.e., *eq('Nancy')*.

Besides query APIs, Gremlin provides a set of update APIs to build a graph database. For example, an empty TinkerGraph can be built using the query *tinkergraph = TinkerGraph.open()*, and then, the Gremlin traversal source *g* can be obtained with the query *g = traversal().withEmbedded(tinkergraph)*. After that, we can use some Gremlin APIs to update the created *tinkergraph*, e.g., *addV()* (adding a vertex to the graph), *addE()* (adding an edge to the graph), and *drop()* (removing elements or properties from the graph).

Note that, SQL-like declarative graph query languages are usually designed for specific GDBs, while the Gremlin query language is widely used in most popular GDBs. In the DB-Engine Ranking of Graph DBMSs [22], 66% (23/35) GDBs support Gremlin APIs. Specially, for the top ten GDBs, eight GDBs support Gremlin APIs. Therefore, we mainly focus on the Gremlin query language and the GDBs using Gremlin to retrieve graph data (also called Gremlin-based GDBs) in this paper. We will discuss how to extend Grand to other graph query languages in Section 6.1.

3 APPROACH

In this paper, we propose Grand, a randomized differential testing approach for automatically finding logic bugs in Gremlin-based GDBs. Figure 5 shows the overview of Grand. Grand includes three

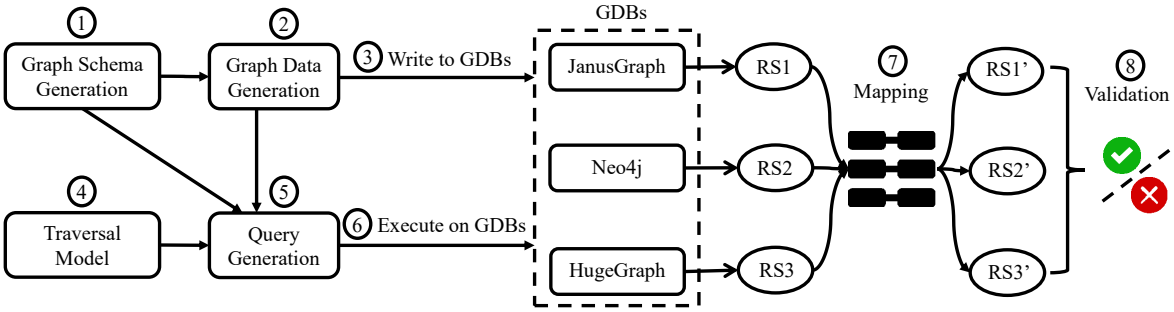

Figure 5: Overview of Grand.

phases, namely graph database generation (Section 3.1), query generation (Section 3.2) and differential testing for GDBs (Section 3.3).

In the phase of graph database generation, Grand first randomly generates the graph schema to define the types of vertices and edges, including labels and properties of vertexes and edges in the graph database (①). Then, the detailed vertices and edges can be randomly generated according to the generated graph schema (②). The generated database will be written into target GDBs (③).

In the phase of Gremlin query generation, we use a model-based query generation approach to generate syntactically correct and valid Gremlin queries. Specifically, we first construct a traversal model for Gremlin APIs (④), and then generate Gremlin queries based on the constructed traversal model and the generated graph database (⑤).

Finally, Grand executes the generated Gremlin queries (⑥) and validates the query results by differential testing. In details, for each target GDB, the result for each query will be recorded and then transformed as a unified query result with the help of the mapping information (⑦). Then, Grand checks these unified results to identify whether there exist discrepancies (⑧). If some GDBs exhibit different outputs, then a potential bug is found.

3.1 Graph Database Generation

Grand generates random database states for target GDBs, i.e., graph schema and graph data, which are used to construct Gremlin queries.

Graph schema generation. A graph schema defines the vertex types and edge types in a labeled property graph. A vertex contains a label and a set of properties. Let $verType =< label, property* >$ denote a vertex type, in which *label* and $property*$ denote its label name and property types, respectively. An edge contains an input vertex, an output vertex, a label and a set of properties. Let $edgeType =< inType, outType, label, property* >$ denote an edge type, in which *inType* and *outType* denote the types of incoming and outgoing vertices, and *label* and $property*$ denote the label name and property types, respectively. Each property contains a property name and its value type. Let $property =< name, type >$ denote a property type, in which *name* and *type* denote the property name and its data type, respectively.

Take the graph in Figure 2 as an example. The type of *vertex* 1 is presented as <person, <name, String>, <gender, String> >. Its label name is person. It has two property types with name and gender, and their data types are both String. The type of *vertex* 2 is presented as <book, <title, String>, <wordCount, Integer> >. Its label name is book. It has two property types with title and wordCount, and their data types are String and Integer respectively. The type for the first edge (from *vertex* 1 to *vertex* 2) is presented as <person, book, write, <publish, Integer> >. Its incoming vertex type is person, and outgoing vertex type is book. Its label name is write, which has a property type publish with data type Integer.

We generate a set of vertex types *VTSet* by the following algorithm. For each vertex type, we first randomly generate its label name, and then generate a set of property types. For each property type, we assign a random data type from Integer, String, Double, Boolean, Float and Long. Note that, we require that all vertex types have different label names, and all property types in a vertex type have different property names, but different vertex types may contain same property types.

We then generate a set of edge types *ETSet* by the following algorithm. To generate an edge type, we first randomly select two vertex types from *VTSet* as its incoming vertex and outgoing vertex, respectively. Then we use the same way to generate its label name and property types as the vertex type generation. Note that, we require that all edge types have different label names, and all property types in an edge type have different property names, but different edge types may contain same property types.

Graph data generation. According to the generated graph schema, Grand further generates the graph data that can be stored in GDBs. Basically, we randomly generate some vertices and edges that conform the graph schema. Then, these generated vertices and edges are written into GDBs. Algorithm 1 illustrates how a vertex and an edge can be generated.

For the vertex generation (Line 1-8), we first randomly get a vertex type *verType* from vertex types *VTSet* generated above. We generate a vertex *vertex* according to the vertex type *verType*. Note that, we randomly select a subset of *verType*'s property types, and generate *vertex*'s properties (Line 5-6). In this way, we can create vertices that lack some properties. For each selected property, we randomly generate its value according to its data type (Line 23).

For the edge generation (Line 9-18), we first randomly get an edge type *edgeType* from edge types *ETSet* generated above. We generate an edge *edge* according to the edge type *edgeType*. The properties for edge generation is the same as that in vertex generation. For the incoming/outgoing vertex of edge *edge*, we randomly

Algorithm 1: Graph Data Generation

```
   Input: VTSet (Vertex types), ETSet (Edge types)
1  Function VertexGeneration() do
2      vertex ← new Vertex();
3      verType ← VTSet.random();
4      vertex.label ← verType.labelName();
5      PTSet ← verType.randomPropType();
6      vertex.properties ← PropGeneration (PTSet);
7      return vertex;
8  end
9  Function EdgeGeneration(vertices) do
10     edge ← new Edge();
11     edgeType ← ETSet.random();
12     edge.label ← edgeType.labelName();
13     PTSet ← edgeType.randomPropType();
14     edge.properties ← PropGeneration (PTSet);
15     edge.incoming ← vertices.random(edgeType.inType);
16     edge.outgoing ← vertices.random(edgeType.outType);
17     return edge;
18 end
19 Function PropGeneration(PTSet) do
20     props ← ∅;
21     foreach pType ∈ PTSet do
22         pName ← pType.name();
23         pValue ← generateConstant(pType.type()));
24         props.add(pName, pValue);
25     end
26     return props;
27 end
```

select one vertex that conform *edge*'s incoming/outgoing vertex type (Line 15-16). Note that, for any two vertices, we only add one edge with the same type.

Note that, the maximal numbers of vertex types, edge types, property types, vertices, edges in a graph database, can be specified by GDB testers. In our experiment, we generate at most 10 vertex types, 20 edge types, 20 property types, 100 vertices, and 200 edges for a graph database.

3.2 Model-Based Query Generation

Random query generation is a commonly-used approach for testing databases [13, 14, 52]. However, the effectiveness of GDB testing heavily relies on the quality of query generation. Gremlin has flexible programming interfaces, which consists of a sequence of (potentially nested) Gremlin API calls acting on a graph traversal source g. Although we can extract grammar constraints among Gremlin APIs with automated tools (e.g., JCrasher [30]), a completely random Gremlin query generation based on the grammar for Gremlin APIs can cause two issues. First, we can easily generate grammatically correct but meaningless queries, e.g., $g.V().V().V()$. Second, from the Gremlin grammar, we cannot know the semantics of Gremlin APIs and their parameters. For example, in $g.V().outE('read')$, we cannot know that the parameter of $outE()$ should be a label. Without these semantics, almost all generated Gremlin queries return empty results, which cannot be used to compare return results in differential testing. These two issues can greatly affect the effectiveness of GDB testing.

To address the above two issues, we first construct a traversal model in Gremlin that can exactly understand Gremlin APIs and their semantics. Based on this traversal model, Grand can link Gremlin APIs correctly and generate syntactically correct and meaningful queries. Further, Grand can utilize these semantics and design various strategies to generate valid Gremlin queries, which can return non-empty query results with a high probability. Note that, we only focus on Gremlin query APIs for query generation, and ignore update APIs that are used to build and update graph databases.

3.2.1 Traversal Model in Gremlin. In this section, we explain the traversal model in general, and then explain Gremlin APIs.

A Gremlin query consists of a sequence of Gremlin API calls, which are correctly linked together. However, there exist constraints about how to link Gremlin APIs. Generally, the input type of a Gremlin API in a query should match the output type of its previous Gremlin API. Otherwise, the Gremlin query will be illegal. For example, for Gremlin API $outE()$ (moving to the outgoing edges), its input should be a vertex set, and its output is an edge set. As such, we can construct a query $g.V().outE()$, but cannot construct $g.E().outE()$, because the output of API $g.E()$ is an edge set not a vertex set.

To construct a syntactically correct and meaningful Gremlin query, we need to build a precise traversal model for Gremlin APIs. By carefully studying Gremlin APIs [18, 19, 51], we build a traversal model to represent the legal transformations among Gremlin query APIs, as shown in Figure 6. Our traversal model contains three types of entities (i.e., $\underline{Vertex}$, $\underline{Edge}$, and $\underline{Value}$), three types of operations (i.e., $\underline{Filter}$, $\underline{Predicate}$, and $\underline{Aggregate}$), and $\underline{Constant}$. Figure 6 formally expresses the transformation of entities and the detailed Gremlin APIs. Based on this traversal model, the legal transformation between adjacent Gremlin APIs can be easily followed. For example, $g.V()$ is used to retrieve all $vertices$ in a graph. We can get $Edges$ with the expression $\underline{Vertex}.[outE() \mid inE() \mid bothE()]$, in which, $outE()$, $inE()$, and $bothE()$ can be linked after $\underline{Vertex}$ to obtain an edge set. In addition, our traversal model can present recursion relations among Gremlin APIs in a concise way. For example, $\underline{Vertex}$ can be followed by $\underline{Filter}$, and $\underline{Filter}$ can use $\underline{Vertex}$ as parameter.

Three entities in the traversal model are used to retrieve detailed graph data in a graph. In detail, $\underline{Vertex}$ represents how to obtain vertices using Gremlin APIs. To retrieve vertices, we can use a start Gremlin API (i.e., $g.V()$), append $\underline{Filter}$ operations on $\underline{Vertex}$, append obtaining vertices operations on $Vertex$ (e.g., $out()$), and append obtaining vertices operations on $Edge$ (e.g., $outV()$). Similarly, $\underline{Edge}$ represents how to obtain edges using Gremlin APIs. We can use a start Gremlin API (i.e., $g.E()$), append $\underline{Filter}$ operations to $\underline{Edge}$, or append obtaining edge operations on $\underline{Vertex}$ (e.g., $inE()$). The $\underline{Value}$ expresses property values in the labeled property graphs. We can get all property values belonging to $\underline{Vertex}$ or $\underline{Edge}$ using Gremlin API $values()$, or get a specific property value by adding property name in $values()$ (e.g., $values('name')$). Besides, we can

$$\underline{Vertex} ::= \underline{g}.V() \parallel \underline{Vertex}.\underline{Filter} \parallel Vertex.[out() \mid in() \mid both()] \parallel Edge.[outV() \mid inV() \mid bothV()]$$
$$\underline{Edge} ::= \underline{g}.E() \parallel \underline{Edge}.\underline{Filter} \parallel \underline{Vertex}.[outE() \mid inE() \mid bothE()]$$
$$\underline{Value} ::= [\underline{Vertex} \mid \underline{Edge}].values() \parallel [\underline{Vertex} \mid \underline{Edge}].properties().values()$$
$$\underline{Filter} ::= has(\underline{Predicate}) \parallel [and|or|not]([\underline{Filter} \mid \underline{Vertex} \mid \underline{Edge}]) \parallel where([\underline{Value}|\underline{Aggregate}].is(\underline{Predicate})) \parallel hasNot() \parallel hasLabel()$$
$$\underline{Predicate} ::= [eq \mid neq \mid lt \mid lte \mid gt \mid gte](\underline{Constant}) \parallel [inside \mid outside \mid between](\underline{Constant},\underline{Constant}) \parallel [not \mid and \mid or](\underline{Predicate})$$
$$\underline{Aggregate} ::= [\underline{Vertex} \mid \underline{Edge}].count() \parallel \underline{Value}.[sum() \mid mean() \mid min() \mid max()]$$
$$\underline{Constant} ::= Integer \parallel String \parallel Double \parallel Boolean \parallel Float \parallel Long$$

Figure 6: Traversal model in Gremlin.

get the property value of a specified vertex or edge through the combined Gremlin APIs *properties*().*values*().

Three operations are used to operate graph data in our traversal model. Specially, $\underline{Filter}$ operation can be used to map entities that satisfy certain filter conditions. We can use *has*(), *where*() or *hasLabel*() API to filter vertices or edges that we require. Specially, we can add some parameters to these APIs. For example, *has*(*key*, *value*) can be used to select those vertices or edges whose value of property *key* is equal to *value*. Another example, we can use *where*($\underline{Value}$.*is*($\underline{Predicate}$)) to construct a nested sub-query, in which a sequence of Gremlin API calls act as a parameter in *where*(). The $\underline{Predicate}$ is used to express predicates, e.g., *eq*(*number*) is used to judge whether the incoming object is equal to *number*. Besides basic APIs like *eq*() and *inside*(), we can also use *not*(), *and*(), or *or*() APIs to connect one or more $\underline{Predicate}$ operations. The $\underline{Aggregate}$ represents a series of operations aggregating vertices, edges, or properties. For instance, *count*() can be used to count the number of vertices or edges, and *sum*() is usually used to compute a sum of property values (e.g., *values*('*age*').*sum*()). Finally, $\underline{Constant}$ acts as API arguments and is used for type definition.

Note that some special cases are not described in the above traversal model for clear presentation. First, the parameters in some Gremlin APIs have been ignored in the traversal model. For example, in *hasNot*() API, a property name should be used as a parameter, and in *hasLabel*() API, a label name is required. Second, some extra constraints are required for $\underline{Aggregate}$. For example, although we use $\underline{Value}$.[*sum*() | *min*()] to present that *sum*() and *min*() APIs can follow a $\underline{Value}$, we need to make sure that *sum*() can only be used when $\underline{Value}$ has the Number type.

3.2.2 Query Generation. Based on the above traversal model, we first generate syntactically correct Gremlin queries, and then generate the parameter values in the generated Gremlin queries.

Query statement construction. Guided by the traversal model in Section 3.2.1, we iteratively generate syntactically correct Gremlin queries step by step. Algorithm 2 outlines this generation process. Initially, *query* is set to a Gremlin graph traversal source *g* (Line 1) and the previous traversal step *preStep* is initialized as `Start`, i.e., the traversal source *g* (Line 2). Grand randomly selects a next Gremlin step, until the maximum query length *MaxLength* is reached or *Exit* condition (e.g., the step type is a Constant value) is satisfied (Line 4-21). Note that, the selected next step must be compatible with *preStep* according to our traversal model. The newly generated step is appended into the end of current *query*.

Algorithm 2: Gremlin Query Generation

```
1  query ← 'g' ;
2  preStep ← Start ;
3  length ← 0 ;
4  while length++ < MaxLength do
5      switch getRandomStep(preStep) do
6          case Map do
7              step ← CreateMap(preStep.type);
8          case Filter do
9              step ← CreateFilter(preStep.type);
10         case Property do
11             step ← CreateProperty(preStep.type);
12         case Aggregate do
13             step ← CreateAggregate(preStep.type);
14         end
15         case Exit do
16             break;
17         end
18     end
19     query ← query + '.' + step.toString() ;
20     preStep ← step ;
21 end
22 return query;
```

Specifically, Grand randomly chooses the next step type based on our traversal model and passes the previous step *preStep* as an argument (Line 5). The `Map` step mainly contains the APIs that are followed by $\underline{Vertex}$ and $\underline{Edge}$ in Figure 6, e.g., *out*(), *outV*() and *outE*(). In Filter step, function *createFilter*() can create the detailed operations depicted in Figure 6, including the *has*($\underline{Predicate}$) API, *hasLabel*() API, etc. The `Property` step constructs the $\underline{Value}$ entity, e.g., *values*(). The `Aggregate` step constructs $\underline{Aggregate}$ operations, e.g., *count*() and *sum*().

We further use Algorithm 3 to introduce the generation of a `Filter` step. Grand can create various $\underline{Filter}$ operations in Figure 6. We first randomly select one $\underline{Filter}$ operation in our traversal model (Line 2). In this algorithm, we only use `HasPredicate` to explain how we generate a `HasPredicate Filter` step (Line 3-11). Other `Filter` steps can be generated similarly. If the input *type* is $\underline{Vertex}$, then a vertex property is randomly selected from the generated

Algorithm 3: Create Filter Operation

```
1  Function CreateFilter(type) do
2      switch getRandomFilter() do
3          case HasPredicate do
4              if type = Vertex then
5                  p ← getRandomVertexProperty();
6              else
7                  p ← getRandomEdgeProperty();
8              end
9              predicate = CreatePredicate(p.type);
10             filter ←< p.name, predicate >;
11             filter.type ← type;
12         case ... do
13             ...
14     end
15     return filter;
16 end
```

vertex properties (Line 4-5). Otherwise, an edge property is randomly selected (Line 6-7). According to the data type of the selected property, we create a `Predicate` operation (Line 9) *predicate*. Now, we can create a `HasPredicate` step based on the property name *p.name* and *predicate* (Line 10). Finally, we set the filter's type as its previous step' type (Line 11). Thus, Algorithm 2 can further generate the following step according to this filter's type.

Generating parameter values for Gremlin APIs. Some Gremlin APIs need certain parameters, e.g., a property name *title* used in $values('title')$, and a constant value 234000 used in $eq(234000)$. Without any guidance, these parameters can be generated randomly, resulting in returning empty query results for almost all generated Gremlin queries. To address this issue, we adopt two strategies for generating these parameter values. First, we randomly choose values based on graph database content, i.e., the graph schema and graph data generated in Section 3.1, to increase the probability of returning a non-empty result. For example, for the parameter that needs a property name, we randomly select a property name (e.g., *titile*) from the graph schema. Second, we use a random function to generate parameter values according to the constant type. For example, for $eq(Long)$ API, we randomly generate a long value 234000 for its parameter. We randomly use these two strategies to increase the probability of returning a non-empty result and avoid overlooking some bugs.

An example for Gremlin query generation. Take the query $g.V().where(values('name').is(eq('Nancy'))).values('gender')$ for Figure 2 as an example. As shown in Figure 7, three traversal steps are randomly generated by Algorithm 2, e.g., a `Map` step ($V()$ in ①), a `Filter` step ($where()$ in ②), and a `Property` step ($values()$ in ⑥). Specially, the `Filter` step includes a nested sub-query, which consists of a `Property` API ($values()$ in ③) and a `Filter` API ($is()$ in ④). Since the output of the `Map` step (①) is Vertex, the property name (i.e., `name`) in the nested sub-query of the `Filter` step (②) is randomly selected from the generated vertex properties or randomly created with our random function. Besides, the property value of the `Predicate` API ($eq()$ in ⑤) and the property name of

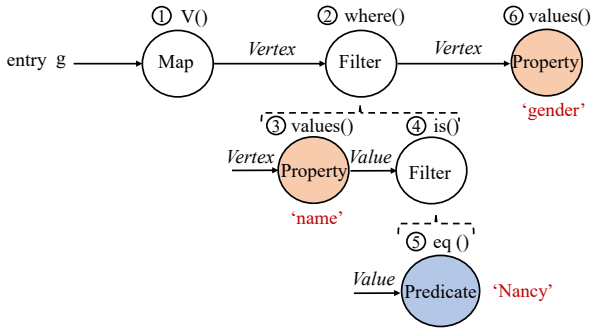

Figure 7: An example for Gremlin query generation.

the Property API ($values()$ in ⑥) can be generated following the same way.

3.3 Differential Testing for GDBs

We use differential testing [45] to detect logic bugs in GDBs. In detail, we first write the same graph data to a group of target GDBs, and then execute the same Gremlin queries on them. By comparing the return results from different GDBs, we identify a discrepancy as a potential bug in these GDBs.

The returned result of a Gremlin query could be a value or a list of vertices, edges or properties. If the returned result is a value or a list of properties, it is straightforward to compare them for different GDBs. When the query result is a list of vertices or edges, the format of the returned results from different GDBs may be different. That is because different ID generation strategies are used in different GDBs. For example, JanusGraph creates an edge *e[7eo-2kg-iz9-268]* to describe an edge, while Neo4j creates an edge *e[2]* to present the same edge. Thus, we need to convert them to a unified format to make the comparison feasible.

To achieve this, we first generate a unique ID *uID* for each vertex and edge generated in Section 3.1. After we write a vertex or an edge into a target GDB, we retrieve its actual ID *actualID* stored in the target GDB. We use a mapping table to record the mapping relations between *uID* and *actualID* in all GDBs for each vertex and edge. For example, as shown in Figure 8, the vertex with ID 1, has different actual IDs 4152, 1605 and 495992707018129408 in JanusGraph, Neo4j and HugeGraph, respectively. By using the mapping table, we can conveniently convert the list of returned vertices or edges from different GDBs into a unified format.

Specifically, for the returned vertices and edges, five steps are needed in our differential testing as shown in Figure 8. For a generated Gremlin query Q, Grand first executes it in the target GDBs (①), and obtains the query results. Then for each GDB, Grand extracts the ID of each vertex or edge in the query results to obtain a list of actual vertex IDs or edge IDs to represent the query result (②). For example, as shown in Figure 8, Grand obtains a vertex ID list [4152, 4136, 4216] for JanusGraph. Next, according to the mapping table (③), Grand converts the real query results to the unified query results (④). Finally, Grand checks whether the unified query results of the target GDBs are the same (⑤). A *True* value will be returned showing that all results for Q are the same, otherwise, a *False* value will be returned to indicate a potential bug.

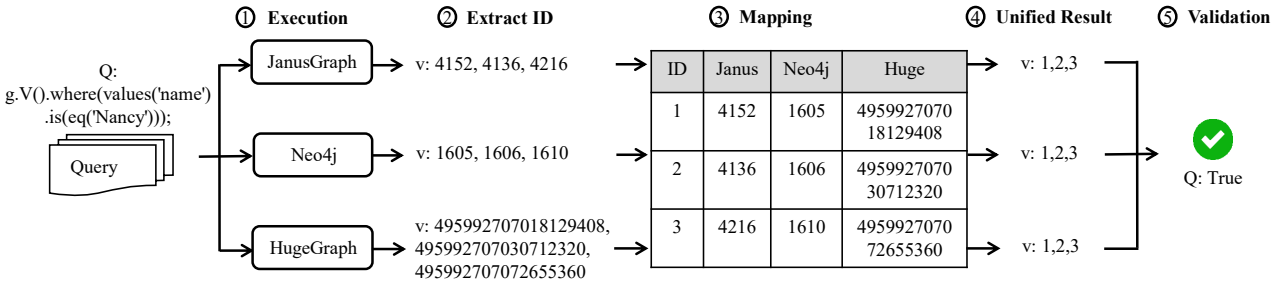

Figure 8: Differential testing in Grand.

4 IMPLEMENTATION

Grand establishes connections to target GDBs before writing the graph into them. For Neo4j, OrientDB, JanusGraph, TinkerGraph and ArcadeDB, Grand uses *GraphTraversalSource* to connect GDB servers remotely, invokes APIs provided by *GraphTraversalSource* to directly write each vertex and edge into GDBs (e.g., $g.addV()$) and appends properties (e.g., $g.V(vertex).property(k, v)$). After that, Grand uses the Gremlin client to submit the generated queries and obtain query results (e.g., $gremlinclient.submit('g.V()')$).

Additional efforts need to be made to write graph data in HugeGraph, as *GraphTraversalSource* is not available in HugeGraph's Java Client. Instead, *HugeClient* [4] is used to connect *HugeGraph Server* by sending HTTP requests. The major difference and effort lie on graph schema creation. Instead of writing the graph directly into GDBs, HugeGraph requires setting up graph schema first. Some extra constraints are explicitly introduced in HugeGraph to standardize graph schema. For instance, the *frequency* of *EdgeLabel* should be set to *multiple*, otherwise only a single edge of each label is allowed between two vertices, which may lead to data overwritten. Beyond that, Grand needs to invoke the specific APIs provided by *HugeClient* to write data (e.g., $hugeclient.graph().addVertex()$), submit queries (e.g., $hugeclient.gremlin().gremlin('g.V()')$), and obtain query results.

For every test round, Grand stores all the graph information in the log files, including graph schema, graph data and ID mapping tables. The executed queries and the results obtained from target GDBs will be stored as well if exceptions or diverse results are detected. Grand provides the method to restore the graph into target GDBs based on these log files, allowing developers re-execute and reproduce the problematic query constantly within its origin graph, in order to confirm the bug and find out the most simplified query statements that can cause the bug.

5 EVALUATION

To demonstrate the effectiveness of Grand, we evaluate Grand on six widely-used GDBs, and detect real-world logic bugs in them.

5.1 Methodology

Target GDBs. We evaluate Grand on six GDBs, i.e., Neo4j [7], OrientDB [11], JanusGraph [6], HugeGraph [3], TinkerGraph [16], and ArcadeDB [9]. Table 1 shows their DB-Engine Ranking of Graph DBMS [22], GitHub starts, and initial release date. We can see that they are among the most popular and widely-used GDBs. Specifically, TinkerGraph was developed by TinkerPop and natively uses Gremlin as its query language. JanusGraph, HugeGraph and ArcadeDB encapsulate Gremlin Server in their own servers, apply some special optimizations, and also natively use Gremlin to query graph data. OrientDB implements its own TinkerPop3 interfaces, and offers OrientDB-TinkerPop3 distribution. We access Neo4j through the Neo4j-Gremlin plugin [8], which is provided by TinkerPop. For all GDBs, we test their *latest release* versions when we start this work, i.e., Neo4j 3.2.3 (with Neo4j-Gremlin 3.4.10, not the latest version), OrientDB 3.2.4, JanusGraph 0.5.3, HugeGraph 0.11.2, TinkerGraph 3.4.10, and ArcadeDB 21.12.1. Note that, we use the latest version of Neo4j-Gremlin, but it does not support the latest Neo4j at that time.

Table 1: The GDBs we tested are popular and widely-used

GDB	Rank	GitHub Stars	Initial Release
Neo4j	1	9.2k	2007
OrientDB	5	4.4k	2010
JanusGraph	8	4.1k	2017
HugeGraph	26	1.7k	2018
TinkerGraph	29	1.4k	2009
ArcadeDB	31	119	2021

Testing methodology. We use a script to configure testing related parameters, e.g., the number of generated queries (1,000 in our experiment), and the graph database generating related parameters, i.e., the max length of each query (10 in our experiment), the max number of vertex types, edge types, vertices and edges (10, 20, 100, 200 in our experiment, respectively).

We configure Grand to run 15 rounds and generate 1,000 random Gremlin queries in each round. At the beginning of each round, Grand first randomly constructs a testing graph database to the target GDBs (Section 3.1), and then applies 1,000 randomly generated Gremlin queries (Section 3.2.2) on the target GDBs to perform a differential testing (Section 3.3). Each reported discrepancy is logged as a potential logic bug. For each bug reported by Grand, we manually reproduce and analyze it, to verify whether it is a real logic bug. Specifically, three steps are needed as follows. First, Grand may generate a complex query to reveal a discrepancy, which makes it challenging to identify its root cause. So, we manually simplify the query to a simple one, which can trigger the same discrepancy.

Table 2: Logic bugs that we found in the tested GDBs

GDB	Detected	Confirmed	Fixed
Neo4j	3	2	1
OrientDB	1	0	0
JanusGraph	3	3	2
HugeGraph	9	9	3
TinkerGraph	3	3	1
ArcadeDB	2	1	0
Total	**21**	**18**	**7**

```
schema.properties("vp0").asDouble().ifNotExist().create();
schema.vertexLabel("vl0").properties("vp0").ifNotExist().
      create();

Vertex v0 = new Vertex("vl0").property("vp0", Double.
      POSITIVE_INFINITY); // Create vertex v[0]
AddVertices(Arrays.asList(v0)) ; // Add vertex v[0]

g.V().has('vl0', 'vp0', Double.POSITIVE_INFINITY);
-- {java.lang.IllegalArgumentException} ✘    {v[0]} ✔
```

Figure 9: HugeGraph cannot process infinity value for Double type.

Second, a discrepancy in Grand cannot tell which GDBs are buggy. So, we manually investigate the expected result of the query in the discrepancy, and then identify which GDBs are buggy. Third, Grand may report different discrepancies for the same bug. We need to understand the root causes of observed discrepancies, and identify whether some discrepancies are caused by the same bug. After filtering out duplicated bugs, we submit an issue for each logic bug to the corresponding community on GitHub.

All experiments were conducted on a cluster with 7 high performance servers, each equipped with a 64-bit CentOS Linux release 8.0.1905 system, a 125GB RAM, and one 20-core 2.50GHz Intel Xeon Gold 5215 CPU. Specifically, each target GDB was installed in one server.

5.2 Overall Bug Detection Results

Grand takes around 160 seconds on average to perform each testing round. For total 15,000 randomly generated Gremlin queries, 6,046 Gremlin queries (40.3%) return non-empty results. We obtain 709 (47 per round) discrepancies, which may be potential bugs. We carefully reproduce and analyze these 709 discrepancies, and find that all of them are true discrepancies. However, after deep analysis, we find that some discrepancies are caused by the same bug.

After carefully analyzing 709 discrepancies, we obtain 21 logic bugs in the six tested Gremlin-based GDBs. Table 2 shows the statistics of logic bugs that we find. We have submitted these bugs to the corresponding community on GitHub. At the time of writing this paper, 18 bugs have been confirmed by developers, and all of them are previously-unknown bugs. Seven bugs have already been fixed by developers.

The 21 bugs have caused severe consequences for these GDBs. Among them, 18 bugs return an unexpected exception for a valid query, whereas they should not. In this case, users cannot get their expected results correctly. Note that, all these bugs return commonly-used exceptions, e.g., IllegalArgumentException, NumberFormatException, NoIndexException and IllegalStateException which can also be returned by invalid queries. The remaining 3 bugs can lead to incorrect query results. In the following, we explain these bugs found in each GDB.

HugeGraph. We found nine logic bugs in HugeGraph. Among them, all nine bugs have been confirmed, and three bugs have been fixed. Specifically, seven bugs were caused due to that HugeGraph cannot support some Gremlin APIs correctly, including *between*(), *outside*(), *not*(), *or*(), *has*(), *hasLabel*() and *order*(). One bug was caused due to that HugeGraph omits boundary values, including infinity value and NaN value. Another one bug was caused by the search constraint of Gremlin APIs that needs to scan the whole table. Even though developers thought users can use alternative queries to avoid the constraint, they still confirmed that they will enhance the usage experience for users after we reported the bug.

JanusGraph. We found three logic bugs in JanusGraph. Among them, all three bugs have been confirmed, two bugs have been fixed by developers, and one of the fixed bugs has been added into their milestone version. In detail, two bugs are related to type coercion. For example, JanusGraph cannot correctly handle numeric comparisons when it comes to the infinity value and NaN value. Unless the data type has been explicitly set in the query, JanusGraph will throw a number format exception. Another bug is related to the unique Query Normal Form (QNF) check, in which JanusGraph cannot necessarily generate a QNF in some cases.

TinkerGraph. We found three logic bugs in TinkerGraph and all of them were confirmed by developers. In one bug, TinkerGraph cannot perform any type coercion with the property values, which means you cannot query the float or double value stored before. This bug has already been fixed. The second bug was caused due to that TinkerGraph cannot compare BigDecimal with Double/Float infinite value, which means you cannot query a property *prop* using *has*() API if an infinity value has been inserted as a *prop* value. Lack of specific function implementation causes the third bug. We found that TinkerPop cannot sort vertices or edges without specifying a property, and developers said they would implement those semantics in subsequent release.

Neo4j. We found three bugs in Neo4j, and all of them were similar to the bugs previously mentioned in TinkerGraph. Actually, this result is not surprising, because the Neo4j-Gremlin we used to query Neo4j is also implemented by TinkerGraph developers. This explains why two different databases meet similar issues.

ArcadeDB. We found two bugs in ArcadeDB, which are similar to the second and third bugs in TinkerGraph.

OrientDB. We found one bug in OrientDB, which is similar to the second bug in TinkerGraph.

5.3 Bug Analysis

We summarize the 21 logic bugs found by Grand into four categories according to their root causes.

Non-robust handling on special values. Five logic bugs are caused by non-robust handling with special values, e.g., Infinity and NaN. For example, HugeGraph cannot process Infinity or NaN value for Double and Float type. In Figure 9, we assign the value

```
1 g.addV('vl0').property('vp0',0.65); // Add vertex v[1011]
2 g.V().has('vp0',0.65);  -- {} ✘    {v[1011]} ✔
```

Figure 10: TinkerGraph returns incorrect results due to lack of type coercion.

```
1 g.addV('vl0').property('vp0', 4);  // Add vertex v[0]
2 g.addV('vl0').property('vp0', 5);  // Add vertex v[1]
3
4 g.V().order().by(asc);
5 -- {org.apache.tinkerpop.gremlin.driver.exception.
       ResponseException} ✘    {v[0], v[1]} ✔
```

Figure 11: TinkerGraph cannot sort vertices when no properties are specified.

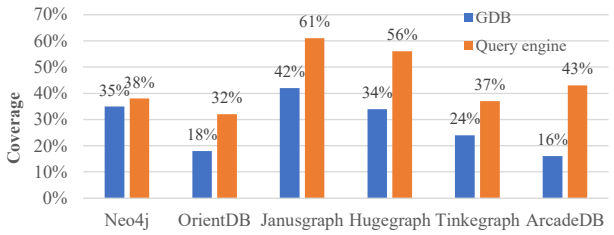

Figure 12: Instruction coverage achieved by Grand.

`Double.POSITIVE_INFINITY` to the property *'vp0'* to create a new vertex `v[0]` (Line 4), and write the vertex `v[0]` to the HugeGraph database (Line 5). We then retrieve the vertex `v[0]` according to its property value `Double.POSITIVE_INFINITY` (Line 7). The expected result is that we can create a new vertex (i.e., `v[0]`) in which the property *'vp0'* is `Double.POSITIVE_INFINITY`. However, an exception `IllegalArgumentException` was thrown. The developers of HugeGraph explained that they ignored Infinity and NaN values previously. Thus, these special numbers were regarded as String values. This bug has already been fixed by HugeGraph developers.

Lack of type coercion. Four logic bugs are caused by lack of type coercion for property values. For example, in TinkerGraph, if users do not explicitly declare double type for property values, they will fail to query expected double values. Figure 10 shows such an example. We first insert a vertex whose property $vp0$ is set as 0.65, and its vertex id is automatically generated as 1011 (Line 1). We then query this vertex by filtering $vp0 = 0.65$ (Line 2). But, this query wrongly returns an empty set, which is different from the results of other GDBs (e.g., HugeGraph). TinkerGraph developers illustrate that TinkerGraph does not perform any type coercion with the property values. This forces users to take care of specific types.

Lack of logic implementation. Eight logic bugs are caused by lack of logic implementation. It is interesting that, except JanusGraph and OrientDB, all the other four tested GDBs cannot sort vertices or edges without specifying a property. Figure 11 shows such an example. TinkerGraph developers stated that they were surprised that it does not work, and thought that they would have had some natural ordering on the element ID.

Incorrect logical implementation. Four logic bugs are caused by incorrect logical implementation in tested GDBs. For instance, HugeGraph does not remove duplicate elements when performing range queries using Gremlin APIs like *not*() and *or*(). Figure 1 shows such an example. HugeGraph developers stated that HugeGraph has not implemented deduplication for overlapping OR operation for now, and they have added this issue into their TODO list to optimize for OR operation.

5.4 Coverage

All the GDBs we tested are implemented in Java. Thus, we utilize JaCoCo [5] to collect instruction coverage for these GDBs, and ran Grand for 24 hours. The detailed instruction coverage is shown in Figure 12. We can see that Grand can achieve instruction coverage from 16% to 42% for these GDBs, and from 32% to 61% for their query engines, which appears to be low. However, this is expected because we mainly focused on testing Gremlin-specific graph queries. We summarize the main reasons as follows. (1) Some GDBs, e.g., ArcadeDB and OrientDB, are multi-model DBMS, and they also support other data models, e.g., document, key/value, time-series, and full-text. We currently did not test these data models. (2) Some GDBs also support various query languages, e.g., ArcadeDB supports SQL, Cypher, and GraphQL. We cannot test the implementation related to these query languages yet. (3) Some GDBs provide features such as user management, replication, transaction and graph analysis, which we did not test.

6 DISCUSSION

In this section, we first discuss how to generalize Grand, the threats to validity, and the limitations of our approach.

6.1 Generalizing Grand to more GDBs

As the first approach to detecting logic bugs in GDBs, Grand can only been applicable to GDBs that support Gremlin APIs. The core philosophy in Gremlin queries is to construct a subgraph through a traversal model, which is used to query GDBs. Grand's design mainly depends on this philosophy. Fortunately, many other query languages also adopt this philosophy to query GDBs. Figure 3 shows such an example for Cypher in Neo4j. Therefore, although some GDBs adopt totally different query languages, we believe, the idea of Grand, e.g., traversal model, model-based query generation and differential testing, can be generalizable to these GDBs.

GDBs that utilize the RDF graph model usually adopt SparQL as their query language. SparQL also needs to construct a subgraph similar to Gremlin. As such, Grand's design philosophy can potentially be adapted to the RDF graph model and SparQL, too.

6.2 Threats to Validity

The main threats to our evaluation are related to the representativeness of our selected GDBs. These six subjects are the most popular open source GDBs that support the Gremlin query language. Neo4j, OrientDB, and TinkerGraph have been developed for more than 10 years, while JanusGraph and HugeGraph are developed actively in the last four years. All of them are well maintained for now. Thus, we think our studied GDBs are representative.

Another threat lies in the manual validation process of logic bugs found in GDBs. We manually reproduce, analyze and deduplicate discrepancies reported by Grand, which may introduce errors due to human error. To make the validation as accurate as possible, three authors analyze all reported discrepancies, and reach consensus for all discrepancies.

6.3 Limitations

Grand has some limitations on finding bugs in Gremlin-based GDBs.

First, Grand mainly focuses on commonly-used Gremlin query APIs. For update APIs (e.g., insert, update and delete graph data), we mainly use them to create databases, and Grand cannot test them systematically. Therefore, logic bugs related to these APIs cannot be found through Grand by now. Actually, we believe Grand can find more bugs when we add more Gremlin APIs in our traversal model.

Second, Grand may miss some bugs due to characteristics of differential testing. Grand cannot detect bugs when the tested GDBs all suffer from the same bugs and return the same wrong results.

Finally, although we can automatically find bugs by repeatedly generating and executing Gremlin queries as well as validating the returned results, we still have to manually find and remove duplicated bugs. In the future, we will investigate how to efficiently filter duplicated bugs reported by Grand.

7 RELATED WORK

In this section, we discuss related works that are close to ours.

7.1 Testing of DBMS

Differential testing of DBMS. Differential testing [45] is commonly used in different domains [29, 31, 38, 52] to find bugs. Its key idea relies on passing the same input to the different testing objects and comparing the output. A difference indicates a potential bug. Slutz first presents RAGS [52] to find bugs in DBMS by differential testing. APOLLO [37] uses differential testing to find performance regression bugs in DBMS. CYNTHIA [53] uses differential testing for testing Object-Relational Mapping systems. Since GDBs adopt different storage structures and query languages, these approaches cannot be easily adopted into GDB testing.

Metamorphic testing of DBMS. Metamorphic testing [44] tests software systems by mutating test cases via metamorphic relations. Rigger et al. provide two metamorphic testing approaches [48, 49], namely Ternary Logic Partitioning (TLP) and Non-optimizing Reference Engine Construction (NoREC), to test DBMS. TLP [49] partitions a query into three sub-queries, and detects bugs by comparing the combination of results of three sub-queries with the result of the original query. NoREC [48] compares the execution results of a given optimized query with its non-optimized version, to detect optimization bugs in DBMS. All the above approaches target relational database systems, and can potentially be applicable on GDBs.

Other testing of DBMS. SQLSmith [14] is a fuzzing tool for testing DBMS. It can randomly generate SQL queries as inputs. SQLancer offers Pivoted Query Synthesis (PQS) [50] approach to find logic bugs by randomly selecting a pivot row as oracle and generating random queries containing the selected row to test DBMS. ADUSA [39] translates SQL query to Alloy specification, generates Alloy instance satisfying query conditions in Alloy specification, and further obtains the expected result from Alloy instance. All the above tools target relational database systems, and cannot be applied on GDBs.

7.2 Performance Testing of GDB

Performance benchmarks for GDB. Angles et al. launch the Linked Data Benchmark Council (LDBC) [26] project to develop benchmarks for graph and RDF data management systems. In the project, LDBC social network benchmark (LDBC SNB) [33] consists of a synthetic social network dataset and three workloads with complex graph dependencies. Pacaci et al. [46] further integrates Kafka into LDBC SNB to simulate real-time graph data. Besides, some benchmarks with simple atomic queries are developed. Angles et al. [27] develop a benchmark consisting of social graphs and atomic queries for social network applications. Lissandrini et al. [42] develop a larger set of queries and operators, and evaluate the performance of GDBs on both synthetic and real graphs. Kovács et al. [41] develop a benchmark to evaluate different GDBs with Wikidata. All these benchmarks are used to measure performance of GDBs, and cannot be used to detect logic bugs in GDBs.

Performance evaluations for GDB. Fernandes et al. [35] analyze five popular GDBs and evaluate flexible graph schemas, query languages, and scalability of GDBs. Abdelaziz et al. [24] perform an evaluation for representative distributed SPARQL graph databases. Wang et al. [55] evaluate the performance of different queries on LDBC SNB benchmark. Besides, TigerGraph also evaluates its own performance using the LDBC SNB benchmark [1]. These works focus on evaluating the performance of GDBs, and do not consider logic bugs in GDBs.

8 CONCLUSION

In this paper, we propose Grand, an automated differential testing approach, for detecting logic bugs in Gremlin-based graph database systems. Specifically, we design a model-based Gremlin query generation approach to generate syntactically correct and valid test cases, and utilize a data mapping approach to obtain uniform query results in different graph database systems. Our experimental results on real-world graph database systems show that Grand is effective in finding logic bugs. Grand finds 21 previously-unknown logic bugs on six widely-used graph database systems. In particular, 18 bugs have been confirmed and 7 bugs have been fixed by developers.

9 DATA AVAILABILITY

The source code of *Grand* is available at Zenodo [23].

ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China (62072444, 61732019), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences (2018142, 201924).

REFERENCES

[1] 2021. *Benchmarking TigerGraph Using the Liked Data Benchmark Council Social Network Benchmark.* Retrieved June 23, 2021 from https://www.tigergraph.com/benchmark/

[2] 2021. *Cypher Query Language.* Retrieved June 23, 2021 from https://neo4j.com/developer/cypher/

[3] 2021. *HugeGraph.* Retrieved August 17, 2021 from https://hugegraph.github.io/hugegraph-doc/

[4] 2021. *HugeGraph-Client Quick Start.* Retrieved August 5, 2021 from https://hugegraph.github.io/hugegraph-doc/quickstart/hugegraph-client.html

[5] 2021. *JaCoCo is a free code coverage library for Java.* Retrieved June 23, 2022 from https://www.jacoco.org/jacoco/

[6] 2021. *JanusGraph.* Retrieved June 23, 2021 from https://janusgraph.org

[7] 2021. *Neo4j.* Retrieved August 5, 2021 from https://neo4j.com/

[8] 2021. *Neo4j-Gremlin.* Retrieved August 5, 2021 from https://github.com/thinkaurelius/neo4j-gremlin-plugin

[9] 2021. *The Next Generation Multi-Model Database Supporting Graphs Key/Value, Documents and Time-Series.* Retrieved June 23, 2022 from https://arcadedb.com/

[10] 2021. *Open Source, Distributed, Scalable, Lightning Fast.* Retrieved June 23, 2021 from https://nebula-graph.io/

[11] 2021. *OrientDB.* Retrieved August 18, 2021 from https://orientdb.org

[12] 2021. *RDF Triple Stores vs. Labeled Property Graphs: What's the Difference?* Retrieved August 5, 2021 from https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/

[13] 2021. *SQLancer.* Retrieved August 5, 2021 from https://github.com/sqlancer/sqlancer

[14] 2021. *SQLsmith.* Retrieved August 5, 2021 from https://github.com/anse1/sqlsmith

[15] 2021. *TigerGraph.* Retrieved September 4, 2021 from https://www.tigergraph.com/

[16] 2021. *TinkerGraph.* Retrieved August 17, 2021 from https://github.com/tinkerpop/blueprints/wiki/tinkergraph

[17] 2021. *TinkerPop.* Retrieved June 23, 2021 from https://tinkerpop.apache.org/

[18] 2021. *TinkerPop Documentation.* Retrieved August 5, 2021 from https://tinkerpop.apache.org/docs/3.4.10/reference/

[19] 2021. *TinkerPop Github.* Retrieved August 5, 2021 from https://github.com/tinkerpop

[20] 2021. *TITAN: Distributed Graph Database.* Retrieved June 23, 2021 from http://titan.thinkaurelius.com/

[21] 2021. *What is openCypher?* Retrieved June 23, 2021 from http://www.opencypher.org/

[22] 2022. *DB-Engines Ranking of Graph DBMS.* Retrieved January 5, 2022 from https://db-engines.com/en/ranking/graph+dbms

[23] 2022. *ISSTA 22 Artifact for "Finding Bugs in Gremlin-Based Graph Database Systems via Randomized Differential Testing".* Retrieved June 21, 2022 from https://doi.org/10.5281/zenodo.6534721

[24] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. 2017. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. *Proc. VLDB Endow.* 10, 13 (2017), 2049–2060.

[25] Ibrahim Abdelaziz, Essam Mansour, Mourad Ouzzani, Ashraf Aboulnaga, and Panos Kalnis. 2017. Query Optimizations over Decentralized RDF Graphs. In *International Conference on Data Engineering (ICDE).* 139–142.

[26] Renzo Angles, Peter A. Boncz, Josep Lluís Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martínez-Bazan, Venelin Kotsev, and Ioan Toma. 2014. The Linked Data Benchmark Council: A Graph and RDF Industry Benchmarking Effort. *ACM SIGMOD Record* 43, 1 (2014), 27–31.

[27] Renzo Angles, Arnau Prat-Pérez, David Dominguez-Sal, and Josep Lluís Larriba-Pey. 2013. Benchmarking Database Systems for Social Network Applications. In *Proceedings of International Workshop on Graph Data Management Experiences and Systems (GRADES).* 15.

[28] Marcelo Arenas, Claudio Gutiérrez, and Juan F. Sequeda. 2021. Querying in the Age of Graph Databases and Knowledge Graphs. In *Proceedings of International Conference on Management of Data (SIGMODE).* 2821–2828.

[29] Shafiul Azam Chowdhury, Soumik Mohian, Sidharth Mehra, Siddhant Gawsane, Taylor T. Johnson, and Christoph Csallner. 2018. Automatically finding bugs in a commercial cyber-physical system development tool chain with SLforge. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018.* 981–992.

[30] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: an automatic robustness tester for Java. *Softw. Pract. Exp.* 34, 11 (2004), 1025–1050.

[31] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *NASA Formal Methods - 4th International Symposium (NFM).* 120–125.

[32] Alin Deutsch. 2018. Querying Graph Databases with the GSQL Query Language. In *Simpósio Brasileiro de Banco de Dados (SBBD).* 313.

[33] Orri Erling, Alex Averbuch, Josep Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD).* 619–630.

[34] Orri Erling and Ivan Mikhailov. 2009. RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media.* 7–24.

[35] Diogo Fernandes and Jorge Bernardino. 2018. Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB. In *Proceedings of International Conference on Data Science, Technology and Applications (DATE).* 373–380.

[36] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of International Conference on Management of Data (SIGMOD).* 1433–1445.

[37] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woon-Hak Kang. 2019. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *Proc. VLDB Endow.* 13, 1 (2019), 57–70.

[38] Timotej Kapus and Cristian Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *International Conference on Automated Software Engineering (ASE).* 590–600.

[39] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O. Laleye, and Sarfraz Khurshid. 2008. Query-Aware Test Generation Using a Relational Constraint Solver. In *International Conference on Automated Software Engineering (ASE).* 238–247.

[40] Norman Köster. 2020. *An Extensible Graph Query Language for Model-Based Information Retrieval in Intelligent Environments.* Ph.D. Dissertation. Bielefeld University, Germany.

[41] Tibor Kovács, Gábor Simon, and Gergely Mezei. 2019. Benchmarking Graph Database Backends - What Works Well with Wikidata? *Acta Cybern.* 24, 1 (2019), 43–60.

[42] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond Macrobenchmarks: Microbenchmark-Based Graph Database Evaluation. *Proc. VLDB Endow.* 12, 4 (2018), 390–403.

[43] Baozhu Liu, Xin Wang, Pengkai Liu, Sizhuo Li, Qiang Fu, and Yunpeng Chai. 2021. UniKG: A Unified Interoperable Knowledge Graph Database System. In *Proceedings of IEEE International Conference on Data Engineering (ICDE).* 2681–2684.

[44] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic testing of Datalog engines. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021.* 639–650.

[45] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107.

[46] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. 2017. Do We Need Specialized Graph Databases? Benchmarking Real-Time Social Networking Applications. In *Proceedings of International Workshop on Graph Data-management Experiences and Systems (GRADES).* 12:1–12:7.

[47] Yuxiang Ren, Hao Zhu, Jiawei Zhang, Peng Dai, and Liefeng Bo. 2021. EnsemFDet: An Ensemble Approach to Fraud Detection based on Bipartite Graph. In *International Conference on Data Engineering (ICDE).* 2039–2044.

[48] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* 1140–1152.

[49] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 211 (2020), 30 pages.

[50] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 667–682.

[51] Marko A. Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language (Invited Talk). In *Proceedings of the Symposium on Database Programming Languages.* 1–10.

[52] Donald S. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of International Conference on Very Large Data Bases (VLDB).* 618–622.

[53] Thodoris Sotiropoulos, Stefanos Chaliasos, Vaggelis Atlidakis, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Data-Oriented Differential Testing of Object-Relational Mapping Systems. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE).* 1535–1547.

[54] Harsh Thakkar, Renzo Angles, Marko Rodriguez, Stephen Mallette, and Jens Lehmann. 2020. Let's Build Bridges, not Walls: SPARQL Querying of TinkerPop Graph Databases with SPARQL-Gremlin. In *Proceedings of IEEE International Conference on Semantic Computing (ICSC).* 408–415.

[55] Ran Wang, Zhengyi Yang, Wenjie Zhang, and Xuemin Lin. 2020. An Empirical Study on Recent Graph Database Systems. In *Proceedings of International Conference on Knowledge Science, Engineering and Management (KSEM).* 328–340.