Testing Graph Database Engines via Query Partitioning

Matteo Kamm
ETH Zurich
Switzerland
matteo.kamm@student.ethz.ch

Manuel Rigger
National University of Singapore
Singapore
rigger@nus.edu.sg

Chengyu Zhang
ETH Zurich
Switzerland
chengyu.zhang@inf.ethz.ch

Zhendong Su
ETH Zurich
Switzerland
zhendong.su@inf.ethz.ch

ABSTRACT

Graph Database Management Systems (GDBMSs) store data as graphs and allow the efficient querying of nodes and their relationships. Logic bugs are bugs that cause a GDBMS to return an incorrect result for a given query (*e.g.*, by returning incorrect nodes or relationships). The impact of such bugs can be severe, as they often go unnoticed. The core insight of this paper is that Query Partitioning, a test oracle that has been proposed to test Relational Database Systems, is applicable to testing GDBMSs as well. The core idea of Query Partitioning is that, given a query, multiple queries are derived whose results can be combined to reconstruct the given query's result. Any discrepancy in the result indicates a logic bug. We have implemented this approach as a practical tool named GDB-Meter and evaluated GDBMeter on three popular GDBMSs and found a total of 40 unique, previously unknown bugs. We consider 14 of them to be logic bugs, the others being error or crash bugs. Overall, 27 of the bugs have been fixed, and 35 confirmed. We compared our approach to the state-of-the-art approach to testing GDBMS, which relies on differential testing; we found that it results in a high number of false alarms, while Query Partitioning reported actual logic bugs without any false alarms. Furthermore, despite the previous efforts in testing Neo4j and JanusGraph, we found 18 additional bugs. The developers appreciate our work and plan to integrate GDBMeter into their testing process. We expect that this simple, yet effective approach and the practical tool will be used to test other GDBMSs.

CCS CONCEPTS

• **Information systems → Database query processing**; • **Software and its engineering → Software verification and validation**.

KEYWORDS

database testing, graph databases, test oracle, automatic testing

1 INTRODUCTION

Graph Database Management Systems (GDBMS) [21, 28, 31] allow storing and querying data as graphs. In recent years, the popularity of such systems has increased drastically due to their applicability in social networks, knowledge graphs [16], and fraud detection [35]. Examples of the most popular GDBMSs are Neo4j [10], JanusGraph [6], RedisGraph [12], and Memgraph [9].

As with any other software, GDBMSs can be affected by various kinds of bugs. A notorious category of bugs are logic bugs, which are bugs that cause the GDBMS to compute an incorrect result. For example, for a given query, a GDBMS might mistakenly omit a vertex from the result set or include an edge that should not be part of the result. Such bugs are difficult to detect by users and might go unnoticed, especially considering the complexity of modern GDBMSs (*e.g.*, Neo4j has 468k LOC).

The state-of-the-art approach to testing GDBMSs, Grand [38], is based on *differential testing* [27]. It generates a test case that is sent to multiple GDBMSs; if the outputs disagree, at least one of the systems is assumed to be affected by a bug. Grand found 21 previously unknown bugs in six GDBMSs, of which 18 bugs were confirmed, 7 were fixed, and 2 were logic bugs. Despite its success in finding bugs, differential testing has major drawbacks in this context. GDBMSs support various query languages that differ in syntax and semantics. Grand was realized for Gremlin, which many, but not all GDBMSs support; for example, RedisGraph is a popular GDBMS that lacks support for Gremlin.[1] Even for Gremlin, there are many differences between different GDBMS implementations; When evaluating Grand, we found that it is prone to false alarms, requiring significant manual effort to analyze potential bug-inducing test cases. As investigated in Section 5.2, for 1,000 randomly generated queries, 615 were considered as potential bugs by Grand; we analyzed a random sample of 30 test cases and found that all of them were false alarms. In the evaluation of the original paper, the authors carefully analyzed 709 test cases exposing differences between GDBMSs, among which they identified only 21 bugs.

Various approaches have been proposed to test Relational Database Management Systems (RDBMS), which can also be affected by logic bugs. The state-of-the-art test oracle for detecting logic bugs

---

[1] See https://github.com/RedisGraph/RedisGraph/issues/274.

is *Query Partitioning* [30], which is based on the idea that from a given query, multiple so-called *partitioning queries* can be derived, each of which computes a part of the original query's result. By combining the partitioning queries' results and comparing the combined result with the original query's result, discrepancies can be located that indicate a bug in the DBMS. As a concrete instantiation of this general idea, *Ternary Logic Partitioning (TLP)* [30] was proposed. The key insight of TLP is that given an original query, three partitioning queries can be derived, each of which contains an additional filtering constraint based on a predicate ϕ. Based on the insight that this predicate ϕ evaluates to either TRUE, FALSE, or NULL for a given context, three filter predicates ϕ, $\neg\phi$, and ϕ IS NULL are applied, one of which should evaluate to TRUE for a given row. TLP has been shown to be effective in testing RDBMS. However, current TLP implementations are not applicable to GDBMS due to the significant difference between RDBMS and GDBMS. Therefore, it is not yet clear whether TLP is still effective in testing GDBMS.

Listing 1: An illustrative example of a logic bug found using Ternary Logic Partitioning in Neo4j.

```
1  CREATE (:L {p:"test"})
2  CREATE INDEX FOR (n:L) ON (n.p)
3
4  MATCH (n:L)
5  RETURN COUNT(n) // c1 = 1
6
7  MATCH (n:L) WHERE n.p STARTS WITH lTrim(n.p)
8  RETURN COUNT(n) // c2 = 0
9
10 MATCH (n:L) WHERE NOT (n.p STARTS WITH lTrim(n.p))
11 RETURN COUNT(n) // c3 = 0
12
13 MATCH (n:L) WHERE (n.p STARTS WITH lTrim(n.p)) IS NULL
14 RETURN COUNT(n) // c4 = 0
15
16 // TLP validates that c1 = c2 + c3 + c4
```

The key insight of this paper is that the high-level idea of Query Partitioning, and specifically TLP, is applicable and effective in finding logic bugs in GDBMSs and addresses the aforementioned challenges. As a *metamorphic testing* approach, TLP checks for inconsistencies within a single system. Thus, TLP can be applied to various GDBMSs that might differ in syntax and semantics. Unlike Grand, TLP does not raise any false alarms. For a fully automated testing approach, TLP must be combined with a test case generation approach. To this end, we propose a simple rule-based generator. The core of the approach is a metamodel generator, which accounts for the lack of schemas in some of the GDBMSs by creating and using an internal schema.

Listing 1 shows an example of a bug that we found in Neo4j 4.6 using TLP. The first two CREATE statements in lines 1 and 2 set up the database state. The first statement creates a new node with label L and property p with value test. The statement in line 2 creates an index on the newly created label-property combination. Lines 4 to 14 contain queries, each of which uses a MATCH clause that counts the number of nodes. The first query in lines 4 to 5 is the original

query, which does not use a filter constraint. The three queries in lines 7 to 14 are the partitioning queries. The first partitioning query calculates the number of nodes with label L where the predicate ϕ evaluates to TRUE, the second where the predicate ϕ evaluates FALSE and the last where the predicate ϕ evaluates to NULL. Since the original query outputs 1 and the subsets are disjoint subsets that partition the initial result, we would expect one of the other three counts to be exactly one as well. In this case, however, all other counts were zero, which indicates a logic bug.

We implemented the approach as a tool called GDBMeter. To evaluate the effectiveness and generality of GDBMeter, we tested the three well-established GDBMSs Neo4j, RedisGraph, and Janus-Graph. We found 40 previously unknown bugs, of which 27 have already been fixed and 14 are logic bugs. By logic bug, we mean the GDBMS gives incorrect results without errors and warnings. Note that Grand cannot be applied to test RedisGraph, as it lacks support for Gremlin. Neo4j and JanusGraph were extensively tested by Grand, which found 3 bugs in each of these GDBMSs, none of which was a logic bug. Despite these efforts, we found and reported 18 additional bugs, 5 of which are logic bugs. In addition, the developers provided positive feedback on our work and plan to integrate GDBMeter into their testing process. We compared GDBMeter to Grand and found that Grand reports a large number of potential bugs, the majority of which are false alarms, while the potential bugs reported by GDBMeter do not have false alarms.

Overall, this paper makes the following contributions:

- It demonstrates how the Query Partitioning test oracle [30], in particular, Ternary Logic Partitioning, can be applied on GDBMSs to find logic bugs.
- It provides a comprehensive evaluation of the oracle on three widely-adopted GDBMS, in which the technique found 40 new bugs.

2 BACKGROUND

Graph Database Management Systems. GDBMSs store and manipulate data as graphs. A directed graph G consists of vertices V and edges E, which we denote as $G = (V, E)$. The set E is a subset of $V \times V$ and we can think of an edge $(v_1, v_2) = e \in E$ as a connection that starts at v_1 and ends at v_2. Note that $(v_1, v_2) \neq (v_2, v_1)$ because these are directed edges for which the order matters. GDBMSs are often schema-less, meaning that data does not have to adhere to a fixed structure. This allows software systems to evolve over time without requiring schema changes and data migrations.

Labeled property graph model. The labeled property graph model is one of two commonly used models in modern GDBMSs [36]. Neo4j, JanusGraph, and RedisGraph are examples of GDBMSs that use the labeled property graph model. This model is a refinement of the pure mathematical model described above. In the labeled property graph model, vertices are commonly referred to as *nodes* and edges as *relationships*. Nodes and relationships can have key-value pairs attached. These pairs are named *properties* and are usually specified using JavaScript Object Notation (JSON). Lastly, labels can be used to mark nodes (relationships). Nodes (relationships) of the same label belong together and form a subset of all the nodes (relationships). Queries typically operate on these label sets for performance reasons. Labels on relationships are also referred to

as relationship types. Contrary to RDBMSs, where data related to the connection of two entities is modeled as an intermediate table, GDBMSs treat edges as first-class citizens, meaning that data can be directly stored as part of an edge itself.
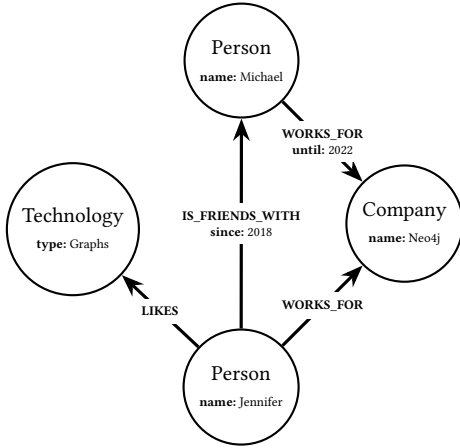

Figure 1: An example of a labeled property graph.

Figure 1 shows an example of a labeled property graph. The graph consists of four nodes and four relationships. The Person node with property name set to "Jennifer" has a relationship IS_FRIENDS_WITH with the Person node named "Michael". This relationship has a property since that specifies since when this relationship exists.

Graph Database Query Languages. Unlike RDBMSs, which commonly support the standardized *Structured Query Language (SQL)*, various common query languages for GDBMSs exist [14, 17]. Some GDBMSs also support multiple languages. The two most prominent ones [36] are Gremlin [32], which is the graph traversal language of Apache TinkerPop [3], and Cypher [22] which was developed for Neo4j [10]. There has been an effort in making Cypher an open standard called openCypher [11]. Neither of those two languages is formally specified and they are therefore subject to change [15].

Cypher. Cypher is a declarative query language that provides a visual way of matching nodes and their relationships. The ASCII-art syntax uses round brackets to represent nodes and arrows for relationships. Listing 2 depicts an example of a Cypher query. It selects all the movies that were directed by the person named "Tom Hanks". The fact that a person directed a movie is represented by a label on the respective relationship.

Gremlin. Gremlin is a functional graph traversal language that composes so-called Gremlin steps. The steps are the primitives of the Gremlin graph traversal machine, which ultimately executes the supplied queries. In total, there are approximately 30 such steps [4]. Listing 3 shows an example of a query written in Gremlin. First, we select all the vertices that have the label "Person" and the property name set to "Tom Hanks". Then we follow all outgoing edges with label "DIRECTED" and finally return all the vertices that we can reach like this which have label "Movie". The traversal-style is

noticeable in this example, since we specify a path through the graph by calling a functional API.

Listing 2: An example of a Cypher query.

```
1  MATCH (:Person {name: "Tom
   ↪  Hanks"})-[:DIRECTED]->(movie:Movie)
2  RETURN movie
```

Listing 3: An example of a Gremlin query.

```
1  g.V()
2   .has("Person", "name", "Tom Hanks")
3   .outE("DIRECTED")
4   .inV()
5   .hasLabel("Movie");
```

Automated Testing. In this paper, we present a new and automated way of testing GDBMSs. Automated testing of GDBMSs consists of two steps. First, an appropriate test case must be generated. For GDBMSs, this refers to statements creating a database as well as a query that is subsequently validated. Various generation approaches have been proposed to test RDBMSs [18, 20, 23]. Importantly, a test oracle is required, which is the mechanism that validates the result of a test case. In this work, we demonstrate how TLP, an oracle originally proposed to test RDBMSs, is applicable to testing GDBMSs.

3 APPROACH

In this paper, we propose an automated testing approach for GDBMSs that we implemented as GDBMeter, a tool that automatically detects bugs in GDBMS. The core of the approach is its test oracle, called Ternary Logic Partitioning (TLP), which was previously proposed to test RDBMSs.

GDBMeter operates in three phases, as shown in Figure 2. In step (1), GDBMeter generates a metamodel that describes available labels for nodes and edges as well as property names and their respective types. Not every GDBMS provides support for schemas, so the metamodel aims at generating coherent data by creating and using an internal schema. Step (2) generates a random graph based on the metamodel. Additionally, create, read, update, and delete (CRUD) statements are randomly generated to create diverse database states, some of which might enable triggering bugs. In step (3), a random query is generated and TLP is applied to validate its result. If a logic bug is detected, GDBMeter reports the bug-inducing test case to the user. Every logic bug reported by TLP is a real bug, that is, no false alarms are reported.

3.1 Metamodel Generation

First, GDBMeter generates a so-called *metamodel*. The metamodel encodes the available graph labels, properties, as well as their types. This model addresses the challenge that some GDBMSs lack support for schemas. Not using a schema would result in a large number of
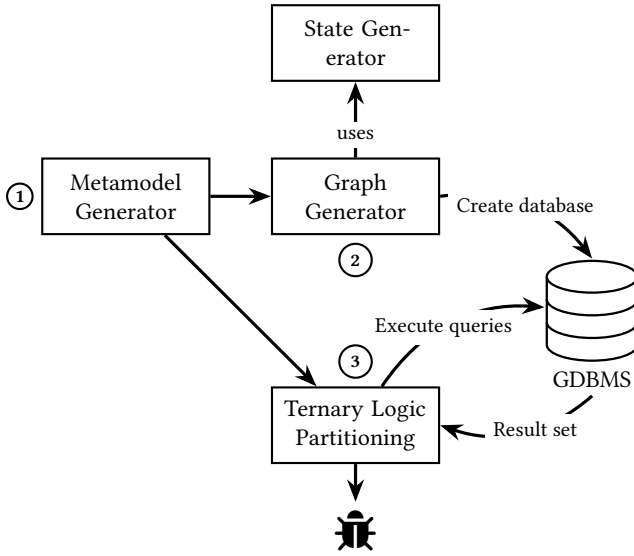
Figure 2: GDBMeter first uses the metamodel generator to create a metamodel, which is then used by the graph generator and the test oracle. The graph generator uses the state generator to generate the actual graph using CRUD operations. Finally, a query is generated and the test oracle is applied. Any bugs that are identified during this step are reported to the user.

meaningless queries as, for example, queries matching for random labels would likely result in empty results.

Let L be the set of valid identifiers for labels and properties for the GDBMS under test. Let T be the types supported by the GDBMS, such as, strings, booleans, integers and points. The structure of our metamodel M is a 4-tuple (L_V, L_E, P_V, P_E), where $L_V, L_E \subseteq N$ describe the available labels for nodes and edges respectively, and P_V, P_E describe the available properties for nodes and edges respectively. L_V, L_E are sets that contain valid, randomly generated, strings. P_V (P_E) is a function of type $L_V \mapsto \mathcal{P}(L \times T)$ ($L_E \mapsto \mathcal{P}(L \times T)$) where $\mathcal{P}$ denotes the power set. This mapping describes which properties (*i.e.*, string-type combinations) can be found on which label.

Consider the graph of Figure 1. It has the metamodel $M = (L_V, L_E, P_V, P_E)$ with the following components:

$$
\begin{aligned}
L_V &= \{\texttt{Person}, \texttt{Company}, \texttt{Technology}\} \\
L_E &= \{\texttt{LIKES}, \texttt{WORKS\_FOR}, \texttt{IS\_FRIENDS\_WITH}\} \\
P_V &= \{(\texttt{Person}, \{(\texttt{name}, \texttt{String})\}), \\
&\quad (\texttt{Company}, \{(\texttt{name}, \texttt{String})\}), \\
&\quad (\texttt{Technology}, \{(\texttt{type}, \texttt{String})\})\} \\
P_E &= \{(\texttt{WORKS\_FOR}, \{(\texttt{until}, \texttt{Date})\}), \\
&\quad (\texttt{IS\_FRIENDS\_WITH}, \{(\texttt{since}, \texttt{Date})\}), \\
&\quad (\texttt{LIKES}, \{\})\}
\end{aligned}
\tag{1}
$$

Note that, since the edge label LIKES lacks properties, we use an empty set to denote its properties. With this information, the schema of a graph is completely described. GDBMeter can, based on this information, generate random graphs or, as is used in some cases, generate a GDBMS-specific schema.

Listing 4: We describe the metamodel using two abstract classes. One class represents the schema, and the other is an entity that can be either a node or an edge. For each entity, we store its associated name as well as the corresponding properties and types. The actual implementation is more complex since we have to support different data types depending on the GDBMS.

```
1  class Schema {
2      Map<String, Entity> nodeSchema;
3      Map<String, Entity> relationshipSchema;
4  }
5
6  class Entity {
7      Map<String, Type> availableProperties;
8  }
```

Listing 4 shows the conceptual components of the metamodel. L_V and L_E are the key sets of the maps in the Schema class. The mappings P_V and P_E are represented by the field `availableProperties` in the `Entity` class. To generate a new metamodel M, GDBMeter first generates L_V and L_E. It generates random, valid, names for the labels. Then, P_V and P_E are created by generating random name-type combinations. For some GDBMSs, the property names have to be unique and this has to be taken into consideration during the metamodel generation.

3.2 Graph Generation

Based on the metamodel M, the graph G is generated. To this end, GDBMeter generates a set of vertices V that adhere to the metamodel M following the label-property mapping. For each of the $|V \times V|$ potential directed edges, we generate an edge with a fixed probability. The edges also adhere to the metamodel M by only generating valid labels and respective properties.

Algorithm 1 describes the graph generation algorithm. Lines 1–9 describe how we generate a set of properties. To do so, we iterate over all elements (name-type combinations) of the schema. For each entry, we generate a random boolean value, if it is true, we include the current property in our subset. The returned value P consists of name-value pairs, where the first component is the property name and the second one is its value.

Lines 10–20 describe how we generate nodes and edges. In both functions, we first sample a random label. Then, based on the selected label, we select the available properties and generate a subset of all the available properties. This is done using the algorithm described above. Finally, the node (edge) is constructed and returned. For the edge construction, u and v describe the outgoing and incoming nodes respectively.

Finally, lines 30–34 describe how the graph can be generated based on the metamodel M. First, n nodes are created using the

---

Algorithm 1 The graph generation algorithm which consists of four different functions.

1: **function** MAKEPROPERTIES(S)
2:     $P \leftarrow \varnothing$
3:     **for all** $(n, t) \in S$ **do**
4:         **if** RandomBoolean() **then**
5:             $P \leftarrow P \cup \{(n, \text{generateValue}(t))\}$
6:         **end if**
7:     **end for**
8:     **return** P
9: **end function**
10: **function** MAKENODE(L_V, P_V)
11:     $l \xleftarrow{R} L_V$
12:     $p \leftarrow \text{makeProperties}(P_V(l))$
13:     **return** Node(l, p)
14: **end function**
15: **function** MAKEEDGE(L_E, P_E, u, v)
16:     $l \xleftarrow{R} L_E$
17:     $p \leftarrow \text{makeProperties}(P_E(l))$
18:     **return** Edge(l, p, u, v)
19: **end function**
20: **function** MAKEGRAPH(M)
21:     $(L_V, L_E, P_V, P_E) \leftarrow M$
22:     $V, E \leftarrow \varnothing$
23:     **loop** n times
24:         $V \leftarrow V \cup \{\text{makeNode}(L_V, P_V)\}$
25:     **end loop**
26:     **for all** $u \in V$ **do**
27:         **for all** $v \in V$ **do**
28:             **if** RandomBoolean() **then**
29:                 $E \leftarrow E \cup \{\text{makeEdge}(L_E, P_E, u, v)\}$
30:             **end if**
31:         **end for**
32:     **end for**
33:     **return** Graph(V, E)
34: **end function**

---

function described before. The number of nodes n can be configured; in our implementation, it is set to a random value between 1 and 6, which we found to work well empirically. Once all the nodes are generated, we loop over all possible edges (*i.e.*, every possible combination of start and end nodes). Then, based on a random variable with a binomial distribution with $p = 0.5$, we generate the edges. Finally, the graph is constructed and returned.

3.3 Database Generation

The database generator is used to generate random statements that manipulate the database state. For each statement kind, we define an empirically-determined interval that describes the number of statements to be generated. GDBMeter iterates over all available kinds of statements and generates a random integer n in the specified interval. This integer n is then used to generate exactly n queries of this kind using a statement generator implementation. The generated statements are executed in a random order in between the create statements of the actual graph. The result of this

random process is a graph database in a deterministic state which is ready to be tested by our test oracles. Since the query languages vary between GDBMS, this component must be GDBMS-specific. Table 1 shows the possible statement kinds for Neo4j.

Many statements require one or multiple expressions. Thus, we use an expression generator. This generator randomly selects applicable operators, functions, and leaf nodes (*i.e.*, variables and constants). Once a maximum depth is reached, only leaf nodes are considered. This ensures that the expressions generated are not excessively large. Constants are generated using a random data generator which is biased to generate boundary values, such as minimum and maximum integers.

3.4 Ternary Logic Partitioning

Ternary Logic Partitioning is an instance of the general partition strategy idea proposed by Rigger and Su [30]. The high-level idea of their approach is that a predicate (*i.e.*, an expression of type boolean used as a filter) on a node or edge must evaluate to TRUE, FALSE or NULL. A given query can therefore be partitioned into three new queries that return disjoint subsets of the original result set. One query selects all nodes and edges where the predicate p holds, one query where p does not hold, and one for which p evaluates to NULL. To implement these three queries, we generate three predicates: p, NOT p, and p IS NULL. Each predicate is then used once in a filter clause to partition the original result set. These predicates are randomly generated.

Some GDBMSs lack support for NULL values. For these, the partitioning involves only two disjoint subsets. Algorithm 2 shows our adapted version of TLP. First, we generate a predicate P and a query Q. Then, the partitioning queries (R, S, T) are generated based on the three predicate variants of P. Finally, we select the nodes using Q as well as R, S, T and validate that these two sets are the same. If they are not, we have detected a bug.

Algorithm 2 The Query Partitioning oracle.

1: **function** QUERYPARTITIONING(M)
2:     $P \leftarrow \text{generateExpression}(M, \text{boolean})$ ▷ P is a predicate
3:     $Q \leftarrow \text{generateSelectionQuery}(M)$
4:     $R \leftarrow \text{modifyFilterClause}(Q, P)$
5:     $S \leftarrow \text{modifyFilterClause}(Q, \neg P)$
6:     ▷ T is only generated if the GDBMS supports NULL values
7:     $T \leftarrow \text{modifyFilterClause}(Q, P \text{ IS NULL})$
8:     **return** SelectNodes(R, S, T) $\overset{!}{=}$ SelectNodes(Q)
9: **end function**

---

4 IMPLEMENTATION

We implemented our approach as a tool called GDBMeter. Currently, it supports testing Neo4j, RedisGraph, and Janusgraph. We implemented it in about 6,000 lines of code (LOC). An implementation of the TLP oracle requires only about 150 LOC. The project is open-source and publicly available at https://github.com/gdbmeter/gdbmeter.

The random statement and query generation of GDBMeter produces syntactically valid statements, which, however, can result

Table 1: The different query kinds of Neo4j supported by GDBMeter and simplified example queries.

Kind (Keyword)	Example
CREATE	`CREATE (:LABEL {property: true})`
CREATE (TEXT) INDEX	`CREATE INDEX name FOR (n:LABEL) ON (n.property)`
DELETE	`MATCH (n:LABEL) DELETE n`
DROP INDEX	`DROP INDEX name IF EXISTS`
REMOVE	`MATCH (n:LABEL) REMOVE n.property`
SET	`MATCH (n:LABEL) SET n.property = false`

Table 2: We tested the most popular GDBMSs. All numbers are the latest as of September 2022.

GDBMS	DB-Engines[2]	GitHub	LOC[3]	First Release
Neo4j	1	10.1k	468k	2007
RedisGraph	-	1.6k	44k	2018
JanusGraph	7	4.6k	93k	2017

in semantic errors when being executed. For instance, a division by zero can result in an error. Such semantic errors are difficult to avoid statically. Thus, we annotated each statement kind with a list of so-called *expected errors* (*i.e.*, errors that happen at run time and are not classified as bugs). If a GDBMS encounters such an expected error during execution, GDBMeter would not report the test case as bug-inducing. Determining whether an error is expected requires domain knowledge. Unlike TLP, which only reports real bugs, omitting an expected error might raise a false alarm.

5 EVALUATION

In our evaluation, we aimed to evaluate the effectiveness of GDB-Meter in finding new bugs in GDBMSs as well as compare it with the state-of-the-art approach Grand [38].

Tested GDBMSs. We considered three GDBMSs, Neo4j, Redisgraph, and JanusGraph. Table 2 demonstrates their popularity and importance based on a widely-used ranking as well as the number of GitHub stars. Neo4j is the most popular, widely-used GDBMS, and the largest GDBMS of the three GDBMSs that we considered. It can be queried using the Cypher query language. Similarly, RedisGraph, an extension of the well-known NoSQL database Redis, uses the Cypher query language. While the RedisGraph developers aim to adhere to the openCypher standard [11], RedisGraph only supports a subset of Cypher features. JanusGraph uses the Tinker-Pop graph computing framework [3] and the underlying graph traversal language Gremlin. Moreover, JanusGraph supports different index backends such as Apache Lucene [2] and Elasticsearch [5]. We tested the latest available versions of these GDBMSs. For Neo4j, we tested versions 4.4.8 and 4.4.9. We tested the development versions of RedisGraph (up to commit `166a643f3`), which is included in version 2.8.19. For JanusGraph, we tested version 0.6.2.

Test environment. We used a 4-core Intel i7-4790K CPU and 16 GB of memory running Arch Linux 5.19 for our bug finding effort. To run GDBMeter, we used Java 11 with the JVM flag `OmitStackTraceInFastThrow`.[4]

5.1 Effectiveness

Study methodology and challenges. We tested the GDBMSs over a period of roughly three months aiming to report unique, previously unknown bugs. Once we identified a potential bug, we reduced the bug-inducing test case to a minimal version [37]. Next, we searched the bug tracker of these GDBMSs to prevent reporting the issue that had already been reported. Finally, if we believed that the bug was likely unknown, we reported it to the issue tracker. To avoid duplicate reports, after reporting a bug, we modified GDBMeter to avoid generating the problematic pattern until the bug was fixed.

Found bugs. Table 3 shows an overview of the bugs and their statuses. Overall, we reported 43 bugs. Of these, we consider 40 bugs as real, previously unknown, unique bugs, 35 of which have been confirmed by the developers. Of the confirmed bugs, 27 bugs have been fixed by the developers. This demonstrates that the majority of our bugs were deemed important by the developers. We reported 2 duplicate bugs; one of these was due to GDBMeter generating two seemingly unrelated test cases, which had the same root cause.

Table 4 shows the types of the bugs that we found. Overall, 14 were logic bugs that we aimed to find. GDBMeter also found a total of 10 crash bugs, which are bugs that caused the server to exit while executing a query. All but one of these bugs were found in RedisGraph. Although Neo4J is written in Java, we considered a bug that causes the application to exit with a stack overflow error as a crash. Some of the crashes were due to illegal memory accesses—RedisGraph is implemented in the C language, which is known to be vulnerable to such bugs. We reported 16 error bugs, that is, bugs that caused an unexpected internal error. Such bugs allowed the GDBMSs to continue processing subsequent queries. Finally, 1 bug that we found caused the GDBMS to hang indefinitely. GDBMeter also found a bug in the Java client for Redis, called Jedis [8], which did not handle the double values for infinity and Not a Number (NaN) correctly.

Developer Feedback. We received highly encouraging feedback on our work. The developers of RedisGraph informed us that they

---

[2]A database ranking based on various factors: https://db-engines.com/en/ranking/graph+dbms

[3]These numbers are best-effort estimates. We calculated them using cloc while excluding tests.

[4]Exception classes that are thrown multiple times are optimized by the JIT compiler in such a way that the stacktrace is removed. This flag prevents this behaviour and ensures proper traceability. For more information on this see: https://github.com/neo4j/neo4j/issues/12874

Table 3: We found 40 previously unknown bugs, 27 of which have been fixed. One bug has been found in Jedis which is not a GDBMS and therefore not listed here.

GDBMS	Fixed	Verified	Closed	
			Intended	Duplicate
Neo4j	10	13	3	2
RedisGraph	15	20	0	0
JanusGraph	1	1	0	0
Total	26	34	3	2

Table 4: Types of the bugs that we found in Neo4J, Redis-Graph, and JanusGraph. The client bug was in Jedis, a Java client for Redis.

GDBMS	Logic	Crash	Error	Hang	Client
Neo4J	5	1	12	0	-
RedisGraph	9	9	2	1	-
JanusGraph	1	0	2	0	-
Total	15	10	16	1	1

are planning to integrate GDBMeter into their testing process to extensively test their database. They stressed the usefulness of GDBMeter with respect to finding clear logic bugs: "*We've seen several academic teams developing tools for finding bugs in graph databases, but most of the time, the queries are generated using fuzzing techniques and seem synthetic. This is not a problem by any means, but we found that researchers couldn't identify the root cause of the issues detected. If I understand correctly, your method has the potential to make this task easier.*"

The developer of Jedis appreciated that we provided a helpful test case that made reproducing the bug easier for them. Most bugs that we reported for RedisGraph were fixed within a few days, which could be an indicator of the importance of the bugs.

5.2 Comparison with Grand

We wanted to compare our approach with Grand [38], the state-of-the-art approach to finding bugs in GDBMSs, which is based on differential testing. Grand has found a total of 21 bugs, 7 of which have been fixed. However, in our initial trial runs, we found that Grand reported many false alarms. This is a severe limitation, as analyzing the potential bug-inducing test cases requires manual effort. We did not find this limitation explicitly mentioned in the paper, which also lacks an evaluation of the false alarm rate.[5] Since TLP only reports real bugs, our main goal was to evaluate the false alarm rate in Grand. As the false alarm rate was prohibitively high, we could not conduct a thorough comparison between the effectiveness of GDBMeter and Grand.

Table 5: A sample of 30 potential bugs that Grand found in 10,000 queries grouped by exception type.

Type	Number
ClassCastException (to Comparable)	11
IllegalArgumentException	8
Parsing Error	6
NumberFormatException	3
IllegalStateException	1
NoIndexException	3

Analysis and results. We executed Grand for 10 iterations, each of which generated 1,000 queries. Grand reported 615 of the 10,000 queries to be potential bugs. We randomly selected 30 potentially bug-inducing test cases, and analyzed them. Differential testing reports only the difference in the output, not the issue or root cause of the issue. Based on our judgment, we classified all of these 30 issues as false alarms. The discrepancies were due to differences in exception handling, as shown in Table 5. For instance, 11 of those 30 reported bugs were due to `ClassCastException` being thrown for HugeGraph and TinkerGraph but not for JanusGraph. In one case, HugeGraph threw an `IllegalStateException`, while the other two GDBMSs simply returned `null`. 6 of the potential bugs were due to illegal symbols triggering different parsing errors.

We reported this issue illustrated on three examples on the project's issue tracker on GitHub in September 2022.[6] The authors responded in the issue that some of these false alarms were due to bugs in their tool, and others are cases that Grand considers as bugs due to the difference in their outputs: "*Some of exceptions are caused by bugs in our tool. For example, we should compare the detailed exception messages Not a legal range: [0, -7248751818768758783] instead of the exception, or maybe we should avoid to generate an odd string value. We will fix them. The third exception is expected in Grand. We think they are bugs due to the different outputs. Actually, they are caused by lack of logic implementation.*"

We further inspected the 21 bugs reported by the Grand authors, which they all classified as logic bugs. Different from previous work [29, 30], which defined logic bugs as bugs that cause an incorrect result to be computed, the Grand authors considered also unexpected errors as logic bugs.[7] We found that 16 bugs were due to internal errors, and 2 issue links referred to pull requests created by Neo4J developers in 2014. We believe that most such internal errors can be found with an implicit test oracle, such as for the error bugs that we found. Only 3 issues were due to differences in the query's result,[8] which we considered as logic bugs in this work. None of these 3 issues has been addressed by code changes; one bug was counted as fixed by the Grand authors due to an update to the documentation.[9]

---

[5] The paper hints that not all issues reported indicate real bugs: "*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.*" Furthermore, the paper mentions that "*After carefully analyzing 709 discrepancies, we obtain 21 logic bugs in the six tested Gremlin-based GDBs.*" without clarifying whether the remaining 688 test cases were false alarms or duplicate bug-inducing test cases.

[6] The issue can be found at https://github.com/choeoe/Grand/issues/1

[7] The paper mentions the following: "*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 errors.*"

[8] See https://github.com/apache/incubator-hugegraph/issues/1586, https://github.com/apache/incubator-hugegraph/issues/1734, and https://issues.apache.org/jira/browse/TINKERPOP-2603

[9] See https://issues.apache.org/jira/browse/TINKERPOP-2603.

6 SELECTED BUGS

This section gives an overview of the interesting bugs we found. Note that this selection is necessarily biased. For brevity, we show only reduced test cases that demonstrate the underlying core problem, rather than the original test cases that found the bugs. The original queries were typically significantly more complex and contained statements irrelevant to reproduce the bugs.

NaN value optimization bug in Neo4j. Listing 5 shows a query that produces the result NaN, false, false. The first value is *Not a Number* (NaN) [1]. A comparison with NaN produces the value false which works as expected in the second expression. The last expression is where the bug occurs. Neo4j incorrectly assumes that it can change NOT(0.0 < (0.0/0.0)) into 0.0 >= (0.0/0.0) which then evaluates to false as any comparison with NaN should. This assumption is incorrect, because NOT(false) is true and, therefore, this is a case of an incorrect optimization. This example shows that handling NaN values correctly can be challenging, especially when combined with query optimizations.

Listing 5: Neo4j incorrectly replaces the logical not operation when an operand is not a number (NaN).

```
1   RETURN (0.0/0.0), 0.0 < (0.0/0.0), NOT(0.0 < (0.0/0.0))
```

Neo4j string comparison bug. Listing 6 shows a set of queries that demonstrate a logic bug. First, a node is created with the property p set to "test". Then we ask for all nodes where property p starts with its lTrim value. lTrim removes leading white spaces from an expression. In our case, it leaves the value unchanged, and "test" STARTS WITH "test" evaluates to true. That is why line 2 returns a count of 1. We then create a normal index on the property p. Finally, we query for the same nodes again, but this time the count is 0. This test case demonstrates incorrect behavior related to indices and the function lTrim.

Listing 6: Neo4j does not return a node that is intended to be part of the result set when an index is present.

```
1   CREATE (:L {p:"test"})
2   MATCH (n:L) WHERE n.p STARTS WITH lTrim(n.p) RETURN
    ↪   COUNT(n)
3   CREATE INDEX FOR (n:L) ON (n.p)
4   MATCH (n:L) WHERE n.p STARTS WITH lTrim(n.p) RETURN
    ↪   COUNT(n)
```

RedisGraph NaN value comparison bugs. Listing 7 shows a collection of comparison queries written for RedisGraph that return incorrect results. Each of them returns the exact negation of the truth value it is supposed to return according to the IEEE Standard for Floating-Point Arithmetic [1]. This example shows that even simple comparisons involving NaN values can be implemented incorrectly. These incorrect results could occur in more complex queries and result in incorrect result sets.

Listing 7: RedisGraph handles comparisons with NaN values incorrectly.

```
1   RETURN 0.0/0.0 = 1
2   RETURN 0.0/0.0 <> 1
3   RETURN 0.0/0.0 <= 1
4   RETURN 0.0/0.0 >= 1
```

RedisGraph distance query results in an infinite loop. Listing 8 shows a bug that is not considered a logic bug but shows that GDB-Meter is also able to detect other interesting bugs. RedisGraph uses RedisSearch [13] as its index backend. To answer certain queries that involve indices, it asks RedisSearch for an answer. In this case, the second query involves the distance, a function which calculates the distance between two points, and since an index is present RedisSearch is consulted. However, since the comparison involves a negative value on one side, RedisSearch runs into an endless loop and never returns. This bug also occurs even when no node is present. This example shows that the bugs found using GDB-Meter have an impact on other projects too (*e.g.*, the ones that use RedisSearch as an index backend).

Listing 8: RedisGraph runs into an infinite loop when comparing a distance to a negative value.

```
1   CREATE INDEX FOR (n:L) ON (n.p)
2
3   MATCH (n:L)
4   WHERE distance(point({ longitude: 1, latitude: 1 }),
    ↪   n.p) <= -1
5   RETURN n
```

RedisGraph null value in WHERE clause bug. Listing 9 shows a logic bug related to null values in WHERE clauses. The expression (null <> false) XOR true evaluates to null because the left side of the XOR is already null. When the WHERE clause is not true, then COUNT(n) should evaluate to zero. However, in this example, Redis-Graph returns a COUNT(n) of one because it incorrectly assumes that the expression is true.

Listing 9: RedisGraph returns a node although the WHERE clause evaluates to null.

```
1   CREATE (:L)
2   MATCH (n:L) WHERE (null <> false) XOR true RETURN
    ↪   COUNT(n)
```

JanusGraph mixed index where one property is not present bug. Listing 10 shows a logic bug related to mixed indices of multiple properties. A mixed index can be used for lookups on any combination of indexed keys and supports multiple condition predicates [7]. Lines 1-3 create an appropriate schema consisting of two properties p and q. We then index those two properties on label L through a

mixed index backend. After creating a node with label L and property p set to 1, we would expect the query of line 10 to return a count of 0 since there is no node with label q. Instead, JanusGraph returns a count of 1 which is incorrect.

Listing 10: JanusGraph returns a node when a mixed index is present, although the condition does not match said node.

```
1  l = makeVertexLabel("L").make()
2  p = makePropertyKey("p").dataType(Integer.class).make()
3  q = makePropertyKey("q").dataType(Integer.class).make()
4
5  buildIndex().addKey(p).addKey(q).indexOnly(l)
6                              .buildMixedIndex();
7
8  g.addV("L").property("p", 1)
9
10 g.V().hasLabel("L").has("q").count() // 0
11 g.V().hasLabel("L").has("q", not(eq(2))).count() // 1
```

7 DISCUSSION

Challenges. One challenge that we faced during the testing of GDBMS, RedisGraph in particular, was that we were unable to reproduce a class of bugs. The bugs appeared to happen sporadically during the execution of seemingly unrelated queries. We then reported the bug by providing the stack trace as well as any other relevant information. Eventually, the developers of RedisGraph were able to find that one bug occurred due to internal locks not being held and a respective invariant being violated.

Limitations. Our testing could use more complex features of the query languages that we support (*i.e.*, Cypher and Gremlin). We have focused on the most important features such as Create, Read, Update and Delete (CRUD) operations as well as indexing features. We found that RedisGraph has some peculiarities when printing floating-point numbers[10] which made it difficult to compare them exactly to our expected result. Because of this, we used an epsilon when comparing floating-point numbers during the execution of our oracle.

8 RELATED WORK

Differential testing of DBMS. Differential testing [27] is a widely-used testing technique that is applicable when multiple systems implement the same behavior for a set of inputs. Its core idea is to pass a common input and if the systems' outputs disagree, a bug in at least one of the systems has been detected. In the context of data-centric systems, this technique was first proposed for testing RDBMSs and implemented as a system called RAGS [33]. Other examples include CYNTHIA [34] for testing Object-Relational Mapping (ORM) systems, DiffStream [25] for testing distributed stream processing systems, and APOLLO [24] for testing for performance regressions of database systems.

Grand [38] realized differential testing for GDBMSs based on the insight that many GDBMSs support the Gremlin language. For

---

test case generation, Grand uses a model-based approach to generate valid Gremlin queries. Besides Grand, GDsmith [26] has been described as an approach to test GDBMSs in a technical report. GDsmith applies differential testing for systems that support the Cypher query language, which is another popular query language. For test case generation, GDsmith uses skeleton generation and completion to generate semantically valid Cypher queries. GDsmith is not publicly available, which is why we could not compare with it. The major drawback of differential testing in this context is that various graph query languages exist, so Grand and GDsmith can only be applied to systems that support Gremlin and Cypher, respectively. Furthermore, even minor differences between the implementation of such query languages cause false alarms, as shown in Section 5.2. TLP addresses these limitations and is applicable to testing a single GDBMSs without reporting false alarms.

Metamorphic testing. Metamorphic testing [19] addresses the test oracle problem by, based on an input and output of a system, deriving a new input and a test oracle that validates the new output by comparing it with the initial output. The approach relies on finding an effective metamorphic relation, which infers the expected results. Ternary Logic Partitioning [30], first proposed for testing RDBMSs, is a metamorphic testing approach. The authors used is to find 175 bugs in widely used RDBMSs. The tool in which they implemented the approach, SQLancer, is highly popular on GitHub and widely used by companies. In this work, we have demonstrated that this technique is also applicable in the context of GDBMSs. TLP's key advantage is that, unlike differential testing, it raises no false alarms. To the best of our knowledge, no other metamorphic testing approaches have been proposed for testing GDBMS.

9 CONCLUSION

This paper has demonstrated that Ternary Logic Partitioning (TLP), a testing approach that was previously proposed for testing RDBMSs, can also be applied to testing GDBMSs. In our evaluation on three widely-used GDBMSs, we have found and reported a total of 43 bugs, 14 of which are logic bugs. Despite Neo4j and JanusGraph having been tested extensively by the state-of-the-art, we found and reported 18 additional bugs in these GDBMSs. Unlike differential testing, TLP avoids false alarms, enabling running it as a fully automated approach. In future work, we expect that this simple, yet effective approach and the practical tool will be used to test other GDBMSs and be integrated into their testing processes.

REFERENCES

[1] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[2] 2022. Apache Lucene. https://lucene.apache.org/. Accessed: September 20, 2022.

[3] 2022. Apache TinkerPop. https://tinkerpop.apache.org/. Accessed: August 10, 2022.

[4] 2022. Apache TinkerPop Documentation. https://tinkerpop.apache.org/docs/current/reference/. Accessed: September 20, 2022.

[5] 2022. ElasticSearch. https://www.elastic.co/elasticsearch/. Accessed: September 20, 2022.

[6] 2022. JanusGraph. https://janusgraph.org/. Accessed: August 2, 2022.

[7] 2022. JanusGraph, Index Management. https://docs.janusgraph.org/schema/index-management/index-performance/. Accessed: September 20, 2022.

[8] 2022. Jedis. https://github.com/redis/jedis. Accessed: September 11, 2022.

[9] 2022. Memgraph. https://memgraph.com/. Accessed: August 2, 2022.

[10] 2022. Neo4J. https://neo4j.com/. Accessed: August 2, 2022.

[11] 2022. openCypher. https://opencypher.org/. Accessed: August 10, 2022.

[12] 2022. RedisGraph. https://redis.io/docs/stack/graph/. Accessed: August 2, 2022.

[13] 2022. RedisSearch. https://github.com/RediSearch/RediSearch. Accessed: September 6, 2022.

[14] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (sep 2017), 40 pages. https://doi.org/10.1145/3104031

[15] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (sep 2017), 40 pages. https://doi.org/10.1145/3104031

[16] Marcelo Arenas, Claudio Gutierrez, and Juan F. Sequeda. 2021. Querying in the Age of Graph Databases and Knowledge Graphs. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, 2821–2828.

[17] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2019. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *CoRR* abs/1910.09017 (2019). arXiv:1910.09017 http://arxiv.org/abs/1910.09017

[18] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. 2007. QAGen: Generating Query-Aware Test Databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China) *(SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 341–352. https://doi.org/10.1145/1247480.1247520

[19] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020).

[20] Claudio de la Riva, María José Suárez-Cabal, and Javier Tuya. 2010. Constraint-Based Test Database Generation for SQL Queries. In *Proceedings of the 5th Workshop on Automation of Software Test* (Cape Town, South Africa) *(AST '10)*. Association for Computing Machinery, New York, NY, USA, 67–74. https://doi.org/10.1145/1808266.1808276

[21] Facebook, Inc. 2021. GraphQL. Working Draft, May. 2021. Online at https://spec.graphql.org/.

[22] 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 the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1433–1445. https://doi.org/10.1145/3183713.3190657

[23] Kenneth Houkjær, Kristian Torp, and Rico Wind. 2006. Simple and realistic data generation. In *Proceedings of the 32nd international conference on Very large data bases*. 1243–1246.

[24] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *Proc. VLDB Endow.* 13, 1 (sep 2019), 57–70. https://doi.org/10.14778/3357377.3357382

[25] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. 2020. DiffStream: Differential Output Testing for Stream Processing Programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 153 (nov 2020), 29 pages. https://doi.org/10.1145/3428221

[26] Wei Lin, Ziyue Hua, Luyao Ren, Zongyang Li, Lu Zhang, and Tao Xie. 2022. GDsmith: Detecting Bugs in Graph Database Engines. https://doi.org/10.48550/ARXIV.2206.08530

[27] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[28] Rob Reagan. 2018. Cosmos DB. In *Web Applications on Azure*. Springer, 187–255.

[29] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1140–1152.

[30] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[31] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data.* " O'Reilly Media, Inc.".

[32] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. ACM. https://doi.org/10.1145/2815072.2815073

[33] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 618–622.

[34] Thodoris Sotiropoulos, Stefanos Chaliasos, Vaggelis Atlidakis, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Data-Oriented Differential Testing of Object-Relational Mapping Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1535–1547. https://doi.org/10.1109/ICSE43902.2021.00137

[35] Sakshi Srivastava and Anil Kumar Singh. 2022. Fraud detection in the distributed graph database. *Cluster Computing* (2022). https://doi.org/10.1007/s10586-022-03540-3

[36] Ran Wang, Zhengyi Yang, Wenjie Zhang, and Xuemin Lin. 2020. An Empirical Study on Recent Graph Database Systems. Springer International Publishing, 328–340.

[37] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why? *SIGSOFT Softw. Eng. Notes* 24, 6 (oct 1999), 253–267. https://doi.org/10.1145/318774.318946

[38] 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* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 302–313. https://doi.org/10.1145/3533767.3534409