Testing Graph Databases using Predicate Partitioning

Master Thesis

Author(s):
Kamm, Matteo

Publication date:
2022

Permanent link:
https://doi.org/10.3929/ethz-b-000573006

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Testing Graph Databases using Predicate Partitioning

Master Thesis

Matteo Kamm

September 25, 2022

Advisors: Prof. Dr. Manuel Rigger, Dr. Chengyu Zhang
Department of Computer Science, ETH Zürich

ABSTRACT

Graph Database Management Systems (GDBMS) store data as part of graph structures and allow the efficient querying of nodes and their relationships. With the rise of social networks and big data, graph databases have gained popularity in recent years. Logic bugs in GDBMS are bugs that manifest by returning an incorrect result for a given query (e.g., by returning incorrect nodes or relationships). The impact of such bugs can be extremely severe since they often go unnoticed. In this paper we apply Ternary Logic Partitioning (TLP), an existing technique used to test Relational Database Management Systems (RDBMS), to GDBMS. Moreover, we refine it into an approach that we call Predicate Partitioning (PP). The core idea behind this approach is that the result set of a given query can be partitioned based on a predicate into disjoint subsets. In our experience, these partitioning queries are very effective in testing filtering clauses of queries. We have extensively tested three popular GDBMS and found a total of 40 previously unknown bugs. We consider 16 of them to be logic bugs and the rest are error or crash bugs. We expect that this simple, yet effective, approach will be used to test other GDBMS.

1 INTRODUCTION

Graph Database Management Systems (GDBMS) [23, 28, 32] allow the storage and efficient querying of data stored in graph structures. In recent years, the popularity of such systems has increased drastically due to their applicability in social networks, knowledge graphs [19] and fraud detection [35]. Examples of the most popular GDBMS are Neo4j [11], JanusGraph [7], RedisGraph [14] (an extension of Redis [13]) and Memgraph [10].

As with any other software, GDBMS can be affected by various kinds of bugs. An important category of bugs are logic bugs, which we define as bugs that cause incorrect result sets. For example, for a given query, a GDBMS might mistakenly omit a vertex from the result set or it includes 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 GDBMS (e.g., Neo4j has 468k LOC).

Logic bugs in GDBMS are notoriously difficult to detect. The following reasons make it difficult to automatically test GDBMS:

(1) Each GDBMS uses a different query language to support data retrieval and insertion. This makes it more cumbersome to test multiple GDBMS at the same time because each system needs an individual implementation.
(2) Some languages, such as the Cypher query language [24] are supported by multiple GDBMS; however, the syntax as well as the semantics of the languages vary. This limits differential testing approaches, which compare the results of a query between multiple GDBMS. Only a common subset of the query language, that also follows the same specification, can be used as part of a testing infrastructure.
(3) Some GDBMS do not support fixed data schemas. Generating valid queries to insert and retrieve data becomes more difficult because of this. Moreover, it can happen that two nodes have the same property `p` which is of different type for each of them. This unpredictable behavior can be remedied during the database generation step.

Existing research focuses on differential testing approaches [27]. Such an approach feeds the same test case input to at least two systems that are supposed to implement the same behavior. If the outputs do not agree, we conclude that at least one of the systems is incorrect. There are two downsides to differential testing. First, it can only be applied to the potentially small common core of two systems. In particular, one cannot test GDBMS with different query languages. Moreover, one cannot detect which system exhibits the incorrect behavior if two outputs differ. This step requires manual effort. Differential testing has been employed to test GDBMS through a tool called Grand [38].

Relational Database Management Systems (RDBMS) also suffer from logic bugs and there has been much research in the area of testing said systems. Several tools, such as RAGS [34], SQLsmith [16] and SQLancer [30], have been proposed and successfully used to discover bugs. However, these tools cannot be directly used on GDBMS to detect logic bugs. This is because they only generate Structured Query Language (SQL) queries which cannot be run on GDBMS. Moreover, RDBMS usually work with a schema and a similar concept does not exist in the world of graph databases.

We combine the insights from existing RDBMS and GDBMS testing approaches to form a new testing approach that solves all of the above challenges. We apply the Ternary Logic Partitioning (TLP) technique [31], which has been successfully used to find logic bugs in RDBMS, to graph databases. The key insight of TLP is that a result set $RS(Q)$ of a query Q can be partitioned into disjoint subsets based on a predicate ϕ. A predicate ϕ evaluates to either `TRUE`, `FALSE` or `NULL` for a given context (nodes, edges and variables). Given a query Q, a filtering clause and the predicate ϕ it is now possible to create three new derived queries with the predicates: ϕ, $\neg\phi$ and ϕ `IS NULL`. These three queries partition $RS(Q)$ into disjoint subsets.

Listing 1: An illustrative example of a logic bug found using Predicate Partitioning (PP) 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)
6
7  MATCH (n:L) WHERE n.p STARTS WITH lTrim(n.p)
8  RETURN COUNT(n)
9
10 MATCH (n:L) WHERE NOT (n.p STARTS WITH lTrim(n.p))
11 RETURN COUNT(n)
12
13 MATCH (n:L) WHERE (n.p STARTS WITH lTrim(n.p)) IS NULL
14 RETURN COUNT(n)
```

Listing 1 shows an example of a bug that we found in Neo4j 4.6 by applying Predicate Partitioning (PP), our test oracle approach. The first two `CREATE` statements in line 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 second line creates an index on the newly created label-property combination. Line 4 to 14 represent the queries of the oracle, each being a `MATCH` that counts the amount of nodes. While the first query calculates the number of nodes with

label L where the predicate ϕ is TRUE, the second one matches all of them where the predicate ϕ is FALSE and the last one does the same but with ϕ being NULL. Since the first count 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.

In this paper we propose a new testing approach for testing GDBMS by using Query Partitioning to tackle the test oracle problem and a test case generation approach inspired by existing RDBMS testing approaches. We implemented this approach as a tool called GDBMeter that supports two important graph query languages, Cypher [24] and Gremlin [33]. Note, however, that any other graph query language could be tested using the same approach.

To evaluate the effectiveness and generality of GDBMeter, we tested the three well-established GDBMS Neo4j, RedisGraph and JanusGraph and we found 40 previously unknown bugs. 25 of which have already been fixed. Neo4j and JanusGraph were extensively tested by Grand [38], a differential testing approach, which found 3 bugs in each GDBMS. We compare GDBMeter to Grand and highlight the key advantages of our approach, the lack of false positives and the fact that GDBMeter can be applied to GDBMS that not only support Gremlin. GDBMeter is available at https://github.com/InverseIntegral/gdbmeter.

Overall, this paper makes the following contributions:

- A randomized approach to generate metamodels and labeled property graphs.
- Application of the Query Partitioning technique [31], in particular Ternary Logic Partitioning, on GDBMS to find logic bugs.
- Comprehensive evaluation of the oracle on three widely-adopted GDBMS, in which the technique found 40 new bugs.
- A comparison to an existing approach based on differential testing implemented in Grand [38].

The remainder of this paper is structured as follows. Section 2 provides necessary background information that is needed to understand our approach. Section 3 describes our approach in more detail. In particular, it explains the graph metamodel generation as well as a testing oracle that detects logic bugs. Selected bugs and how they were found is part of section 4. Section 5 evaluates our approach and is followed by a brief discussion of said approach in section 6. Section 7 goes over the related work and section 8 concludes the paper.

2 BACKGROUND

Graph Database Management Systems. Graph Database Management Systems (GDBMS) store and manipulate data as part of graphs. In the mathematical sense, a directed graph G consists of vertices V and edges E, we usually write $G = (V, E)$. 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. This is the mathematical notation used to describe graphs.

GDBMS 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 having to think about the schema changes and data migrations. Instead of SQL, GDBMS use domain specific query languages with a syntax that allows for easy selection of vertices and their corresponding edges.

Labeled property graph model. The labeled property graph model is one of two commonly used models in modern GDBMS [36]. Neo4j, JanusGraph and RedisGraph are examples of GDBMS that use the labeled property graph model. This model is a refinement of the pure mathematical model described above. From now on we also refer to vertices as nodes and edges as relationships because this is the notation used in the labeled property graph model. Nodes and relationships are the core components of this model. Furthermore, nodes and relationships can have key-value pairs attached. These pairs are named properties and are usually written in 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 usually operate on these label sets for performance reasons. Sometimes labels on relationships are also referred to as relationship types. Contrary to Relational Database Management Systems, where data related to the connection of two entities has to be modeled as an intermediate table. GDBMS treat edges as first-class citizens, meaning that data can be directly stored as part of an edge itself.

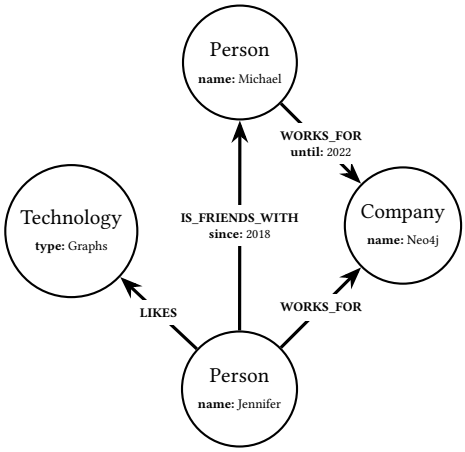

Figure 1: An example of a labeled property Graph.

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 again a property since that specifies since when this relationship exists.

Graph Database Query Languages. Unlike RDBMS, which support the standardized Structured Query Language (SQL), GDBMS do not have one common query language. Instead, each GDBMS supports its own query language, luckily for us, there are some languages that are supported by multiple GDBMS. The two most prominent ones [36] are Gremlin [33], which is the graph traversal

language of Apache TinkerPop [4], and Cypher [24] which was developed for Neo4j [11]. There has been an effort in making Cypher an open standard called openCypher [12]. Neither of those two languages is formally specified and they are therefore subject to change [18].

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. As shown with this example, we do not specify how the data should be fetched but rather which data we want. Fetching the data efficiently is the responsibility of the GDBMS.

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 [5]. 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 very 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.

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

Listing 3: An example of a Gremlin query.

```
g.V()
 .has("Person", "name", "Tom Hanks")
 .outE("DIRECTED")
 .inV()
 .hasLabel("Movie");
```

Automated Testing. In this paper we present a new and automated way of testing GDBMS. Automated testing of databases consist of two steps. First, an appropriate database has to be generated. This challenge has been widely studied for relational databases [20, 22, 25] but at the same time there has been little to no research in the area of graph database generation. We believe that the same ideas can be applied to graph databases. Our implementation is loosely based on the database and expression generator of SQLancer [30]. However, this is not the main focus of our work.

Importantly, a test oracle is required, which is the mechanism that validates the result of a test case. Our work focuses on the class of metamorphic test oracles [21]. These oracles can be used to generate new test cases based on a metamorphic relation and previously known inputs and outputs of a system. An example for such a relation for a program that calculates the sine value is $\sin(x) = \sin(x + 2\pi)$. One can generate a follow-up test case to establish that the relation holds based on the result of a previous test case.

3 APPROACH

In this paper, we propose a metamorphic testing approach, implemented in GDBMeter, a tool that automatically detects bugs in GDBMS. GDBMeter operates on three phases which are depicted in Figure 2. First, the metamodel, an abstract representation of available labels and properties (name-type combinations), is generated (Section 3.1). Then, a random graph is generated based on that model (Section 3.2). Finally, the chosen test oracle is executed on the random graph (Section 3.4). Since syntax and semantics of query languages vary widely between GDBMS, GDBMeter contains database specific components. For instance, the graph and expression generators are considered database specific. Both are implemented manually and in a straight-forward way. GDBMeter also consists of database independent components that take care of data generation, logging and configuration handling.

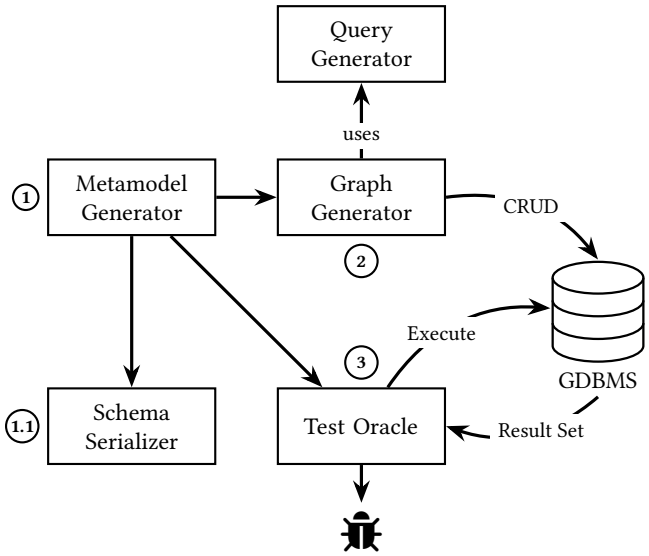

Figure 2: GDBMeter first uses the metamodel generator to create a fresh metamodel, which is then used by the graph generator and the test oracle. Optionally, the metamodel can be exported to the GDBMS if it supports a schema. The graph generator uses the query generator to generate the actual graph using CRUD operations. Finally the test oracle executes queries and reports bugs if any are found.

In the first step, GDBMeter has to randomly generate a metamodel that describes available labels for nodes and edges as well as property names and their respective type (1). This is because not every GDBMS supports schemas and we do need some form of predetermined structure later on during the test oracle phase. Optionally, the metamodel can be passed to the GDBMS by adding corresponding schema constraints (1.1). The second step generates a random graph based on the metamodel (2). To do this create,

read, update and delete (CRUD) queries are randomly generated by consulting the query generator. Lastly, a test oracle is executed on the GDBMS (3). An oracle can be either of type read-only or read-write. The former never modifies any data whereas the latter might delete nodes and edges. If this step detects a (logic) bug, it reports it to the user.

3.1 Metamodel Generation

The first step in the execution of GDBMeter is to generate a so-called metamodel. The metamodel encodes the available graph labels, properties as well as their types. This model is necessary because some GDBMS do not support schemas. However, at runtime we need to know which properties have which type. This is required to generate correct expression and therefore queries. To ensure, with a high probability, that our queries do not return empty result sets, we have to know the labels that appear in our graph.

Let L be the set of valid identifiers for labels and properties for our GDBMS under consideration. And let T be the types supported by the GDBMS, these usually include 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 simply 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 (string-type combinations) can be found on which label.

To illustrate the metamodel, consider the graph graph of Figure 1. It has the metamodel $M = (L_V, L_E, P_V, P_E)$ that looks like this:

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

Note that since the edge label `LIKES` does never have any 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.

In Listing 4 a simplified version of our metamodel in Java code can be seen. 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 fresh metamodel M, we start by generating L_V and L_E. This is done by simply generating random, yet valid, names for our labels. Then P_V and P_E are created by generating random name-type combinations. Depending on the GDBMS under test different types are valid, this is solved through Java generics in our approach. Furthermore, for some GDBMS the property names have to be unique and this has to be taken into consideration during the metamodel generation.

Listing 4: We represent the metamodel in Java by using two classes. One represents the complete schema and one an entity which can be either a node or an edge. For each entity we store its associated name and 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 }
```

3.2 Graph Generation

Based on the just generated metamodel M, the graph G is generated. To do this, we first generate a set of vertices V that adheres to the metamodel M by 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 follow the metamodel M by only generating valid labels and respective properties. Currently, GDBMeter allows exactly one label per node and relationship. In the future it would be possible to drop this restriction.

Algorithm 1 describes the graph generation algorithm. Line 1-9 describe how we generate a set of properties. To do so, we go over all elements (name-type combinations) of the schema. For each entry we generate a random boolean value and if said value 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.

Line 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 node respectively.

Finally, line 30-34 describe how the graph can be generated based on the metamodel M. First n nodes are created using the function described before. The number of nodes n can be configured but is currently set to a random value between 1 and 6. 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 Query Generation

The query generator is used to generate random mutation queries of different kinds. For each kind we define an interval that describes the amount of queries that can be generated. Then we iterate over all the available kinds and generate a random integer n in its interval. This integer n is then used to generate exactly n queries of this kind using a query generator implementation. Since the query languages vary between GDBMS, this component must be GDBMS-specific.

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

```
1:  function makeProperties(S)
2:      P ← ∅
3:      for all (n, t) ∈ S do
4:          if RandomBoolean() then
5:              P ← P ∪ {(n, generateValue(t))}
6:          end if
7:      end for
8:      return P
9:  end function
10: function makeNode(L_V, P_V)
11:     l ←R L_V
12:     p ← makeProperties(P_V(l))
13:     return Node(l, p)
14: end function
15: function makeEdge(L_E, P_E, u, v)
16:     l ←R L_E
17:     p ← 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) ← M
22:     V, E ← ∅
23:     loop n times
24:         V ← V ∪ {makeNode(L_V, P_V)}
25:     end loop
26:     for all u ∈ V do
27:         for all v ∈ V do
28:             if RandomBoolean() then
29:                 E ← E ∪ {makeEdge(L_E, P_E, u, v)}
30:             end if
31:         end for
32:     end for
33:     return Graph(V, E)
34: end function
```

For Neo4j we have listed the possible query kinds in Table 1. Note that the example queries are simplified and are usually much larger. The generated queries 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.

To generate a query we internally use an expression generator which operates in a top-down fashion. It 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 do not become too large. Constants are generated using a random data generator which is biased to generate boundary values, such as minimum and maximum integers. This random data generator is an adapted version of SQLancer's [30] random data generator.

3.4 Predicate Partitioning (PP) Oracle

Ternary Logic Partitioning is an instance of the general partition strategy idea invented by Rigger and Su. The high level idea of their approach is that a predicate (expressions of type boolean) 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 just like the queries.

We present a generalized approach called Predicate Partitioning (PP) that is based on TLP. The core idea stays the same, however, certain GDBMS do not support `NULL` values in their database and in that case, the partitioning involves only two disjoint subsets. To implement PP we used Algorithm 2 which does exactly what is described above. First, we generate a predicate P and a query Q. Then, the partitioning queries (R, S, T) are generated by changing the predicates based on modified versions of P. Finally, we select the nodes using Q as well as R, S, T and we expect these two sets to be the same. If they are not, we know that we detected a bug.

Algorithm 2 The Predicate Partitioning (PP) oracle.

```
1: function PredicatePartitioning(M)
2:     P ← generateExpression(M, boolean)    ▷ P is a predicate
3:     Q ← generateSelectionQuery(M)
4:     R ← modifyFilterClause(Q, P)
5:     S ← modifyFilterClause(Q, ¬P)
6:     ▷ T is only necessary if the GDBMS supports NULL values
7:     T ← modifyFilterClause(Q, P IS NULL)
8:     return SelectNodes(R, S, T) ≟! SelectNodes(Q)
9: end function
```

3.5 Bug Reporting

Whenever the oracle detects a test case that fails it does report said case to the console. GDBMeter does not provide a reduced test case, meaning that some queries might be superfluous when reproducing the bug or even that parts of a query are unnecessary. The reduction step has to be done manually by the developer. Usually, this can be done by minimizing the amount of queries that still trigger the bug and then minimizing the queries themselves. The result of this process is a minimal example that can be reported to the developers of the GDBMS. To ease this tedious manual process, GDBMeter provides a feature that runs queries from a file and reproduces the exact state based on a serialized version of the metamodel. In the future this replay future could be extended to automatically reduce a set of queries to a minimal set that still produces the same error by using the Delta Debugging approach [37]. One project that already applies this approach successfully to reduce C/C++ files is C-Reduce [29].

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`

The random query generation of GDBMeter can sometimes produce expected errors (i.e., errors that happen at runtime that are not classified as bugs). For instance, a division by zero could result in an error but before we run the query we cannot know if said query will result in a division by zero. Therefore, GDBMeter supports expected exceptions that can be specified on a per-query level. If the GDBMS throws an exception for such an annotated query, GDBMeter would not report an incorrect bug.

4 SELECTED BUGS

This section gives an overview over interesting bugs that we found. For brevity, we show only reduced test cases that demonstrate the underlying core problem, rather than the original queries that found the bugs. The original queries are usually much more complex and contain lots of random data that, after reduction, turns out to be irrelevant to reproduce the bug.

NaN value optimization bug in Neo4j. Listing 5 shows a query that produces the result `NaN, false, false`. The first value Not a Number (`NaN`) is defined by the IEEE Standard for Floating-Point Arithmetic [1] and exhibits completely normal behavior. By this standard, a comparison with `NaN` produces the value `false` which is exactly what happens with 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, though, because `NOT(false)` is `true` and therefore this is a case of premature 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 contains 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 untouched and clearly `"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`, note that this is not a string index. Finally, we ask for the exact same nodes again but this time the count is 0. This shows that there is some form of incorrect behavior related to normal (non-text) 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 go wrong when not handled with great care. 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 GDBMeter is also able to detect other interesting bugs. RedisGraph uses RedisSearch [15] 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. Interestingly enough, this bug occurs even when no node is present.. This example shows that the bugs found using GDBMeter 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`. Usually, whenever the `WHERE` clause it not `true` then `COUNT(n)` is zero. However, in this example RedisGraph 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 [8]. 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).buildMixedIndex();
6
7  g.addV("L").property("p", 1)
8
9  g.V().hasLabel("L").has("q").count() // 0
10 g.V().hasLabel("L").has("q", not(eq(2))).count() // 1
```

5 EVALUATION

We evaluated the effectiveness of the Predicate Partitioning oracle for finding logic bugs in GDBMS. Furthermore, we compare our approach to an existing one that uses differential testing and is implemented in a tool called Grand [38].

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

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

Implementation. We implemented our approach in GDBMeter, a framework that is able to test various GDBMS. The whole project, including the infrastructure, is implemented in about 6'000 LOC. An implementation of the partition oracle, however, only requires about 150 LOC. The project is open-source and available at https://github.com/InverseIntegral/gdbmeter.

Tested GDBMS. In our evaluation we considered three popular and widely-used GDBMS (See Table 2). Neo4j is considered to be the industry standard when it comes to graph databases. It can be queried using the Cypher query language. Similarly, RedisGraph, an extension of the well-known NoSQL database Redis, also uses the Cypher query language. However, RedisGraph does only support a subset of Cypher features but they try to follow the openCypher standard [12] where possible. The last graph database that we tested is JanusGraph which uses the TinkerPop graph computing framework [4] and the underlying graph traversal language Gremlin. Moreover, JanusGraph supports different index backends such as Apache Lucene [3] and Elasticsearch [6]. For all GDBMS we tested their latest available versions. In particular, for Neo4j we tested the versions 4.4.8 and 4.4.9. RedisGraph was tested using a self-built version based on the master branch (up to commit `166a643f3`) which is included in version 2.8.19 and for JanusGraph we tested version 0.6.2.

Test environment. We used an 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` which prevents the JIT Compiler from optimizing away exception messages[3].

5.1 Effectiveness

Study methodology and challenges. We tested the GDBMS over a period of roughly three months. Oftentimes we could not proceed with finding bugs effectively because bugs either prevented other bugs from happening or our tool found duplicate bugs that we had already reported. To remedy the latter, we tried to prevent the bug from being found again. To this end, we implemented a feature that could be toggled through a simple configuration that would make sure that a class of bugs did not happen repeatedly. As described earlier, we reduced the test cases manually and made sure that these were actual bugs and not just false positives. We then made sure that no one else found the same bug previously and finally reported it through the issue trackers provided by the developers.

[1] A database ranking based on various factors: https://db-engines.com/en/ranking/graph+dbms
[2] These numbers are best effort estimates. They are calculated using cloc and tests are excluded
[3] For more information on this see: https://github.com/neo4j/neo4j/issues/12874

Table 3: We found 40 previously unknown bugs, 25 of which have been fixed. One bug has been found in Jedis which is not a GDBMS and therefore not listed here.

			Closed	
GDBMS	Fixed	Verified	Intended	Duplicate
Neo4j	9	12	1	2
RedisGraph	14	21	1	0
JanusGraph	1	1	0	0

Oracle implementation. We first implemented the predicate partitioning oracle for Neo4j in a straight-forward manner. After seeing its initial success, we decided to test other GDBMS. In order to reuse as much of our implementation as possible, we chose to target a GDBMS that uses the Cypher query language. RedisGraph was an optimal candidate since it supports similar features as Neo4j, uses Cypher too and is under active development. Finally, we decided to show that our approach is applicable not only to Cypher based GDBMS by implementing the oracle for JanusGraph, which supports the Gremlin query language. This is also one of the advantages over differential testing. The challenge in that was that Gremlin does not support arbitrary predicates like Cypher does, thus, we had to use predefined predicate functions through the Gremlin API.

Found bugs. Table 3 shows the bugs that we found grouped by GDBMS and status. We have found a total of 40 bugs which were previously unknown, 36 of which have already been fixed or confirmed by the developers. This demonstrates that the majority of our bugs are deemed to be important by the developers. Two of our reported bugs were duplicates, one such duplication occurred because GDBMeter generated two seemingly unrelated test cases which turned out to have the same root cause. All other confirmed bugs were unique and we achieved this by thoroughly checking the issue trackers before reporting our bugs. Interestingly enough, GDBMeter also found a total of 9 crash bugs (i.e., bugs that make the server shut down during the execution a query). All of these bugs were found in RedisGraph and some of them had to do with memory management, in particular, illegal memory accesses. Neo4j and JanusGraph, on the other hand, seem to catch exceptions on a per-query basis which allows them to recover even when executing malicious queries. Finally, two bugs that we found caused the GDBMS to hang indefinitely and the execution of the query had to be terminated manually. GDBMeter also found a bug in the Java client for Redis, called Jedis [9], which did not handle the double values for infinity and Not a Number (NaN) correctly.

Soundness. Our approach is sound (i.e., a reported bug is always a real bug). This is because GDBMeter only reports a bug when a query returns an incorrect result, when the GDBMS throws an unexpected exception or when it unexpectedly crashes. We encountered one bug that was marked as "won't fix" for now because the change would be too severe and would require a change in semantics. This bug is an actual bug in our eyes since the database returns an incorrect result, the developers referred to the undefined semantics of this particular case though.

Table 4: A sample of 30 potential bugs that Grand found in 10'000 queries grouped by exception type.

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

5.2 Comparison with Grand

One existing approach that tests GDBMS is Grand [38]. It uses randomized differential testing to find bugs in GDBMS that support the Gremlin query language [33]. The basic idea is that they generate random graphs and queries and execute those on multiple different GDBMS with the same exact capabilities. They then compare the results of those queries and if they do not match, Grand reports a bug. The reasoning behind this approach is that one would expect that different implementations of the same query language follow the same semantics. The crucial component in this process is the model-based query generator. It should support a wide variety of interesting queries while still adhering to the syntax of the query languages. Grand found a total of 21 bugs, 7 of which have been fixed.

We have tried Grand and planned on doing a thorough analysis of its bug finding capabilities. To do this we followed the instruction given on GitHub and we used the provided scripts to perform differential testing between JanusGraph [7], TinkerGraph [17] and HugeGraph [2]. Once we concluded our initial experiments with Grand, we realized that it would be close to impossible to compare our approaches due to the amount of bugs that Grand reported. We suspect that most of them are false positives which would need to be investigated manually. To support this claim, we let Grand run for 10 iterations each of which generated 1'000 queries. 615 of all the 10'000 queries were potential bugs reported by Grand. We inspected 30 of those potential bugs and all of them turned out to be false positives. The inspected bugs occurred due to differences in exception handling. 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` whilst the other two GDBMS simply returned `null`. And 6 of the potential bugs were due to illegal symbols triggering different parsing errors. We have counted the appearances of the different exceptions in those 30 potential bugs and summarized them in Table 4. This strongly implies that there are a lot of false positives. We have reported this on GitHub through an issue[4]. A direct quantitative comparison between Grand and GDBMeter seems infeasible because of this.

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

6 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. Later on the developers of RedisGraph figured out that the bug occurred due to internal locks not being held and a respective invariant being violated at that point. This shows that GDBMeter is able to detect bugs that are notoriously difficult to reproduce because they depend on exact timing and the execution of multiple threads.

Bug importance. It is difficult to quantify the importance of bugs but the feedback that we received of the developers was always positive. The developers of RedisGraph are planning to integrate GDBMeter into their tool chain to extensively test their database. We have been in direct contact with them and they wrote: "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 were really happy 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 for the importance of the bugs.

Generalization to more GDBMS. GDBMeter has been written such that extending it to support more GDBMS would be easily possible. To this end, we separated GDBMS-specific components from general purpose components to enable reuse when implementing support for a new GDBMS. For instance, the graph generation can be reused as long as the GDBMS is based on the property graph model. Testing Cypher based GDBMS is even simpler since there are already complete components which can be used without any modifications. Supporting other GDBMS is also not a problem as can be seen with JanusGraph which uses the Gremlin query language.

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[5] which made it difficult to compare them exactly to our expected result. Because of this we use an epsilon when comparing floating-point numbers during the execution of our oracle. Finally, one limiting factor is the manual reduction that is required when finding a test case. One has to manually remove queries or part of the queries to arrive at a minimal example. And this process requires great care as to not lose any relevant information.

[5]For more information see: https://github.com/RedisGraph/RedisGraph/issues/2417

7 RELATED WORK

Differential testing of DBMS. Differential testing [27] is a testing technique where multiple systems are supposed to implement equivalent behavior, a single input is sent to all of them and if the outputs disagree, a bug in at least one of them has been detected. Slutz applied this technique for testing RDBMS and realized a system called RAGS. GDsmith [26] and Grand [38] are two tools that successfully apply the same idea to GDBMS. GDsmith uses skeleton generation and completion to generate semantically valid Cypher queries which are then sent to different graph database instances. Grand, on the other hand, uses a model-based approach to generate valid Gremlin queries and then uses differential testing to detect logic bugs.

Metamorphic testing of DBMS. Metamorphic testing [21] addresses the test oracle problem by generating new input, for which the result is already known, based on previous input. The metamorphic relation, which infers the expected results, is vital to this approach. Rigger and Su apply this technique to find logic bugs in RDBMS and implemented their approach in a tool called SQLancer. To the best of our knowledge, there is no previous research in the area of metamorphic testing of GDBMS.

8 CONCLUSION

This paper has presented how the idea of Ternary Logic Partitioning (TLP) can be applied to Graph Database Management Systems (GDBMS). In this process, we developed a new form of Query Partitioning that we call Predicate Partitioning (PP). The basic idea behind this approach is to partition the result set of a query based on a random predicate, which is has a domain of only two (sometimes three) values. With this we are able to predict if a query returns an incorrect result whenever a filter step based on a predicate occurs. Through our evaluation on three widely-used GDBMS we have shown that PP is highly effective. It has found a total of 42 bugs, 16 of which are logic bugs. We are convinced that the idea of Query Partitioning can be extended further in the domain of graph databases and that there are more bugs to be found.

ACKNOWLEDGMENTS

We want to thank the developers of the GDBMS for verifying and addressing our bug reports as well as their feedback to our work. Furthermore, we are grateful for the feedback received by the members of the AST Lab at ETH Zürich.

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 HugeGraph. https://hugegraph.apache.org/. Accessed: September 20, 2022.

[3] 2022. Apache Lucene. https://lucene.apache.org/. Accessed: September 20, 2022.

[4] 2022. Apache TinkerPop. https://tinkerpop.apache.org/. Accessed: August 10, 2022.

[5] 2022. Apache TinkerPop Documentation. https://tinkerpop.apache.org/docs/current/reference/. Accessed: September 20, 2022.

[6] 2022. ElasticSearch. https://www.elastic.co/elasticsearch/. Accessed: September 20, 2022.

[7] 2022. JanusGraph. https://janusgraph.org/. Accessed: August 2, 2022.

[8] 2022. JanusGraph, Index Management. https://docs.janusgraph.org/schema/index-management/index-performance/. Accessed: September 20, 2022.

[9] 2022. Jedis. https://github.com/redis/jedis. Accessed: September 11, 2022.

[10] 2022. Memgraph. https://memgraph.com/. Accessed: August 2, 2022.

[11] 2022. Neo4J. https://neo4j.com/. Accessed: August 2, 2022.

[12] 2022. openCypher. https://opencypher.org/. Accessed: August 10, 2022.

[13] 2022. Redis. https://redis.io/. Accessed: September 20, 2022.

[14] 2022. RedisGraph. https://redis.io/docs/stack/graph/. Accessed: August 2, 2022.

[15] 2022. RedisSearch. https://github.com/RediSearch/RediSearch. Accessed: September 6, 2022.

[16] 2022. SQLsmith. https://github.com/anse1/sqlsmith. Accessed: August 8, 2022.

[17] 2022. TinkerGraph. https://github.com/tinkerpop/blueprints/wiki/tinkergraph. Accessed: September 20, 2022.

[18] 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

[19] 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.

[20] 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

[21] 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).

[22] 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

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

[24] 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

[25] 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.

[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] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. *SIGPLAN Not.* 47, 6 (jun 2012), 335–346. https://doi.org/10.1145/2345156.2254104

[30] Manuel Rigger. 2022. SQLancer. https://github.com/sqlancer/sqlancer. Accessed: August 8, 2022.

[31] 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.

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

[33] 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

[34] 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.

[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