Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation

Yuancheng Jiang
National University of Singapore
Singapore
yuancheng@comp.nus.edu.sg

Jiahao Liu
National University of Singapore
Singapore
jiahao99@comp.nus.edu.sg

Jinsheng Ba
National University of Singapore
Singapore
bajinsheng@u.nus.edu

Roland H.C. Yap
National University of Singapore
Singapore
ryap@comp.nus.edu.sg

Zhenkai Liang
National University of Singapore
Singapore
liangzk@comp.nus.edu.sg

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

Abstract

Graph Database Management Systems (GDBMSs) store graphs as data. They are used naturally in applications such as social networks, recommendation systems and program analysis. However, they can be affected by logic bugs, which cause the GDBMSs to compute incorrect results and subsequently affect the applications relying on them. In this work, we propose injective and surjective Graph Query Transformation (GQT) to detect logic bugs in GDBMSs. Given a query Q, we derive a mutated query Q', so that either their result sets are: (i) semantically equivalent; or (ii) variant based on the mutation to be either a subset or superset of each other. When the expected relationship between the results does not hold, a logic bug in the GDBMS is detected. The key insight to mutate Q is that the graph pattern in graph queries enables systematic query transformations derived from injective and surjective mappings of the directed edge sets between Q and Q'. We implemented injective and surjective Graph Query Transformation (GQT) as a tool called GraphGenie and evaluated it on 6 popular and mature GDBMSs. GraphGenie has found 25 unknown bugs, comprising 16 logic bugs, 3 internal errors, and 6 performance issues. Our results demonstrate the practicality and effectiveness of GraphGenie in detecting logic bugs in GDBMSs which has the potential for improving the reliability of applications relying on these GDBMSs.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Database and storage security**.

Keywords

Graph Databases, Logic Bugs, Metamorphic Testing

ACM Reference Format:
Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland H.C. Yap, Zhenkai Liang, and Manuel Rigger. 2024. Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3623307

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3623307

1 Introduction

Graph Database Management Systems (GDBMSs) [2] are designed for storing and querying graph data. They have been rapidly gaining popularity and are becoming increasingly prevalent in various domains. Specifically, GDBMSs operate on vertices and edges as data to support graph storing and matching, which greatly facilitates the usability and efficiency of many applications like social networks [16, 51], recommendation systems [3, 40, 54], and program analysis [28, 29, 52]. According to graph database market statistics [32], the global market size of graph database valued at USD 2.9 billion in 2023 is expected to grow to USD 7.3 billion by 2028 at a compound annual growth rate of 20.2%. This upward trend is driven by many factors, including the rising demand for online schema environments and the real-time big data mining.

Given that GDBMSs are complex software systems with complex algorithms, they are susceptible to logic bugs—resulting in an incorrect result for a given query. Unlike crash bugs, logic bugs in GDBMSs can silently compute incorrect results, which often go unnoticed by both users and developers, making it difficult to fix such bugs. For example, the AgensGraph [7] GDBMS had a logic bug[1] that produced wrong results when counting nodes with a cyclic path (*e.g.,* `MATCH (n)-[]-(n) RETURN COUNT(n)`), which was introduced by a previous patch to remove unnecessary joins but forgot to consider self-joins.

Automatically detecting logic bugs in GDBMSs is challenging due to the difficulties in establishing an effective *test oracle*, which is a mechanism for determining whether a test case has passed or failed [6]. One approach is to use differential testing [27, 55] to compare the results from different GDBMSs, and any discrepancy in the results suggests a potential bug. However, we have observed that even the most basic queries like `MATCH ()-[r]-() RETURN COUNT(r)` can yield different results in different GDBMSs (*e.g.,* Neo4j [37] and MemGraph [34]), due to inconsistent but intended designs, leading to false alarms in differential testing. Another approach is to leverage metamorphic testing to generate mutated graph queries and validate that their results adhere to a prior expectation. In GDBMSs,

[1] https://github.com/bitnine-oss/agensgraph/issues/595

Listing 1: Logic Bug We Found in RedisGraph

```
Q: MATCH (s1:B)-[*1..2]-(s3:A) RETURN count(s1);
   // Result: 204    Response Time: 0.76ms
Q(=): MATCH (s3:A)-[*1..2]-(s1:B) RETURN count(s1);
   // Result: 238    Response Time: 0.75ms
```

the only existing work [22] is based on Predicate Partitioning [42], which aims to generate three disjoint subset queries via Ternary Logic Partitioning [43]. While Predicate Partitioning has been successful in uncovering many bugs in GDBMSs, it is limited to testing the correct handling of the predicate portion of the graph query, ignoring the more important graph patterns part of graph queries. Compared to the *Predicate*, the graph query pattern (*e.g.*, (N)-[]-(N) or ()-[R]-()), which we call *G-Pattern*, is the essential component of graph queries. We recognize the importance of the *G-Pattern* and are motivated to explore novel solutions to improve the limitations of existing works on GDBMS.

In this work, we present a new testing approach called injective and surjective Graph Query Transformation (GQT) to effectively detect logic bugs in GDBMSs. The central idea behind our approach is that *G-Pattern* in graph queries allows for generating follow-up queries via systematic query transformations derived from injective and surjective mappings among directed edge sets of *G-Pattern*. Follow-up queries are designed to compute results that relate to the initial queries' results in a specific way, and any discrepancy indicates a logic bug. Our approach considers two classes of *G-Pattern* mappings of directed edge sets, that is, (1) bijective mappings and (2) injective-only or surjective-only mappings. Using the bijective edge mappings, we generate a follow-up query Q' based on a query Q such that $RS(Q')$=$RS(Q)$, which means their results are equal. To derive the follow-up query Q', we introduce an operator $\textcircled{=}$ that randomly applies a query equivalent transformation. Similarly, using the injective-only or surjective-only edge mappings, we generate a follow-up query Q' based on a query Q such that $RS(Q') \supseteq RS(Q)$ or $RS(Q') \subseteq RS(Q)$, which means the follow-up query has larger or smaller result (at least equal) than the base query. To derive such comparative relations, we introduce variant operators $\textcircled{>}$ and $\textcircled{<}$ that randomly apply a query generalization or restriction transformation. Our approach creates graph query transformations based on the injective and surjective directed edge mappings for three operators (*i.e.,* $\textcircled{=}$, $\textcircled{>}$, and $\textcircled{<}$), which we use to generate test cases from a base query.

Listing 1 demonstrates one logic bug[2] we found using GQT in RedisGraph [39]. The bug-inducing test case consists of one base query Q and one of its equivalent queries $Q^{\textcircled{=}}$ constructed by mutating the graph pattern. The graph pattern in the mutated query is represented in the opposite order preserving the bijective mapping between directed edge sets, which allows for an alternative query giving the same result. As the pair of semantically equivalent unexpectedly output inconsistent results, we identified a logic bug in RedisGraph, which was acknowledged and fixed by the developers.

To assess the effectiveness of our technique, we implemented it as a tool called GraphGenie, which supports two popular graph query languages (*Cypher* and *Gremlin*). We tested GraphGenie in

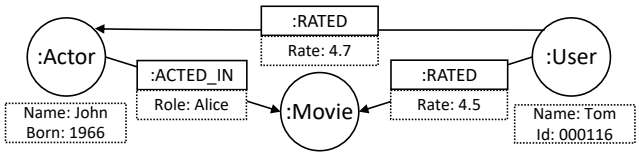

Figure 1: Example of Labeled Property Graph Model

six mature GDBMSs: Neo4j [37], RedisGraph [39], AgensGraph [7], TinkerPop [4], JanusGraph [19], and HugeGraph [18]. We discovered a total of 25 previous unknown bugs consisting of 16 logic bugs (6 fixed and 3 confirmed), 3 internal errors (all fixed), and 6 performance issues (all received positive feedback). Furthermore, we compared GraphGenie with two state-of-the-art approaches based on differential testing [33] and query partitioning [42], implemented as tools named Grand [55] and GDBMeter [22]. In comparison to Grand, which we found to have a false alarm rate of over 80%, our proposed approach is free of false alarms. In comparison to GDBMeter, our approach allowed us to identify 6 logic bugs that GDBMeter was unable to find. We believe that our results are encouraging, suggesting that GraphGenie might become a practical tool for testing GDBMSs.

In summary, we make the following contributions:

- We propose injective and surjective Graph Query Transformation (GQT), a novel and effective approach for identifying logic bugs in GDBMSs. It creates scalable graph transformations on common query elements, generates follow-up queries based on directed edge mappings among *G-Pattern*, using three operators (*i.e.,* $\textcircled{=}$, $\textcircled{>}$, $\textcircled{<}$) to create test oracles to uncover logic bugs.
- We have developed a practical tool called GraphGenie[3] based on our approach. This tool is capable of generating valid base queries, combining rules to generate mutated queries, and identifying logic bugs without any false alarms and supports the top two graph query languages.
- GraphGenie has found various logic bugs in mature GDBMSs. Specifically, we discovered a total of 25 previous unknown bugs, including 16 logic bugs with 6 fixed and 3 confirmed. In addition, GraphGenie also uncovered several internal errors and severe performance issues. GraphGenie's bug reports were positively received and acknowledged by the developers.

2 Background

Graph Database Management System. Graph Database Management Systems (GDBMSs) use various graph models for data representation. This paper focuses on testing the labeled property graph model shown in Figure 1—the most widely used model in GDBMSs such as Neo4j [37], RedisGraph [39], and TinkerPop [4]. These state-of-the-art GDBMSs are large and complex (*e.g.,* more than one million Lines of Code (LoC) in the latest release of Neo4j).

Graph Query Language. Graph query languages (*e.g., Cypher* [35] and *Gremlin* [47]) are used to interact with GDBMSs. We mainly illustrate our approach in the *Cypher* query language. A graph query in Cypher consists of the following elements: (i) *Match* specifies that a graph pattern needs to be matched (in this paper, we do not deal with add, delete and update queries); (ii) graph pattern,

[2] https://github.com/RedisGraph/RedisGraph/issues/2865

[3] GraphGenie is available at https://github.com/YuanchengJiang/GraphGenie

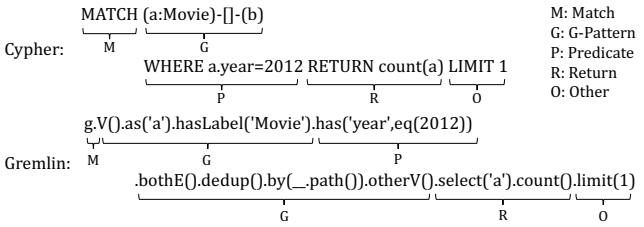

Figure 2: Graph Query Elements

which we call *G-Pattern*, specifies the pattern to be matched with nodes and edges; (iii) *Predicate* specifies a filtering condition on the edges matching the query and typically node and edge labels and properties are used; (iv) *Return* specifies the result and can include aggregation functions such as COUNT(); and (v) *Others* specifies addition constraints on the result set such as ORDER BY, LIMIT, and SKIP. In a query, nodes are written as (VAR:LABEL) and edges are written as [VAR:TYPE]. The VAR represents an optional variable for nodes and edges, which can have a set of LABELS/TYPES associated with it. Nodes and edges are connected via hyphens (-) and arrows that indicate the edge directions (> or <). To match a general graph pattern, *Cypher* allows nodes and edges to be represented without variable names and labels (*e.g.,* () and []). Cypher also allows for more complex graph patterns whose length is variable (*e.g.,* [*1..2] matches graph patterns with the edge length from 1 to 2).

Various graph query languages differ in their syntax and semantics. Figure 2 shows equivalent queries written in *Cypher* and *Gremlin*. However, they typically share common graph query components that provide common functionality. Thus, query transformations can be designed to be general to support testing various graph query languages.

Injection, Surjection, and Bijection. Injections, surjections, and bijections [50] relate how functions map their *domain* (function argument) to the *co-domain* (function result). In an injection, it indicates that the function is a one-to-one mapping, that is, each element of the codomain is mapped to by at most one element of the domain. Surjection indicates the mapping as "onto", if each element of the codomain is mapped to by at least one element of the domain. Bijection is a function that is both injective and subjective.

3 Graph Pattern Mapping

To facilitate the systematic testing of GDBMSs, we find inspiration from injection, surjection, and bijection concepts. In this context, we consider the base query Q as the *domain* and the mutated query Q' as the *co-domain*. Next, we explain how we utilize these concepts to guide the mutation of graph patterns within graph queries.

Graph queries are used to retrieve certain graph patterns from the GDBMSs. Graph patterns represent the data to be retrieved as nodes and edges, which we call *G-Pattern*. As previous works have investigated testing based on the *Predicate* in the WHERE clause,

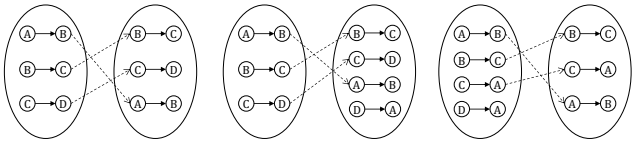

Figure 3: Bijection, Injection, Surjection in Directed Edge Sets

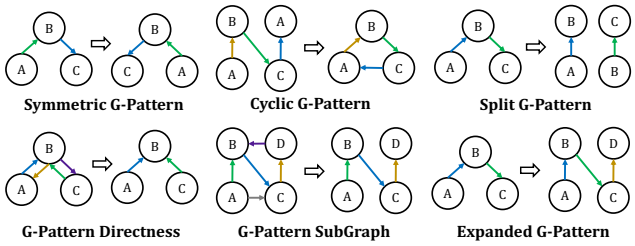

Figure 4: Examples of Graph Pattern Mappings

we propose new testing solutions based on *G-Pattern*, which is one of the essential elements of graph queries. We convert *G-Pattern* to directed edge sets $GP=\{\overrightarrow{e_1}, \ldots, \overrightarrow{e_i}, \ldots, \overrightarrow{e_N}\}$, $\overrightarrow{e_i}=(\mu, \nu)$ where N is the number of directed edges in *G-Pattern* and μ,ν are start and end nodes. An undirected edge e_i in a *G-Pattern* is turned into two oppositely directed edges (*i.e.,* (μ, ν) and (ν, μ)), and we use the shorthand $\overrightarrow{AB}$ to represent a directed edge with start node A and end node B.

Figure 3 demonstrates bijection, injection, and surjection mappings between two directed edge sets. For example, the injective-only mapping in Figure 3 represent the mappings between two graph patterns (GP_1 (A)-->(B)-->(C)-->(D) and GP_2 (A)-->(B)-->(C)-->(D)-->(A)): every directed edge in GP_1 is mapped to GP_2 but unmatched edge exists in GP_2 (non-surjective). We categorize the *G-Pattern* relationship into three classes (*i.e.,* equivalence, restriction, and generalization) by judging the mapping (*i.e.,* $f : GP_1 \mapsto GP_2$) between their directed edge sets.

G-Pattern Equivalence (=) is a bijective mapping between two sets of directed edges from two *G-Patterns* in graph queries. This means that every edge in GP_1 corresponds to a unique and matching edge in GP_2, and no edge is left unmatched, making the two *G-Patterns* equivalent. Examples of *G-Pattern* equivalence include symmetric *G-Pattern*, cyclic *G-Pattern*, and split *G-Pattern* as shown in Figure 4. Specifically, the symmetric *G-Pattern* hold the same directed edge sets with a different order (*e.g.,* $GP_1=\{\overrightarrow{AB}, \overrightarrow{BC}\}$ and $GP_2=\{\overrightarrow{BC}, \overrightarrow{AB}\}$); the cyclic *G-Pattern* share one equivalent backward edge (*e.g.,* $GP_1=\{\overrightarrow{AB}, \overrightarrow{BC}, \overrightarrow{CA}\}$ and $GP_2=\{\overrightarrow{AB}, \overrightarrow{BC}, \overrightarrow{CD}\}$ where A=D); the split *G-Pattern* divides edge sets into disjoint subsets (*e.g.,* $GP_1=\{\overrightarrow{AB}, \overrightarrow{BC}\}$ and $GP_2=\{\{\overrightarrow{AB}\} \cup \{\overrightarrow{BC}\}\}$). To summarize, *G-Pattern* equivalence alters the directed edge set while preserving the bijective mapping.

G-Pattern Restriction ($\leq$) is a mapping between two sets of edges that is surjective-only and non-injective. This means that not every edge in *G-Pattern* GP_1 maps to *G-Pattern* GP_2. Examples of *G-Pattern* restriction include directed *G-Pattern* and subgraph *G-Pattern* as shown in Figure 4. Specifically, the *G-Pattern* directness alters the undirected *G-Pattern* by deleting one direction (*e.g.,* $GP_1=\{\overrightarrow{AB}, \overrightarrow{BA}, \overrightarrow{BC}, \overrightarrow{CB}\}$ and $GP_2=\{\overrightarrow{AB}, \overrightarrow{BC}\}$); the *G-Pattern* subgraph could be generated via the spanning subgraph by deleting edges or the induced subgraph by deleting nodes with a subset directed edge set (*e.g.,* $GP_1=\{\overrightarrow{AB}, \overrightarrow{BC}, \overrightarrow{CD}, \overrightarrow{AC}, \overrightarrow{DB}\}$ and $GP_2=\{\overrightarrow{AB}, \overrightarrow{BC}, \overrightarrow{CD}\}$). *G-Pattern* restriction alters the directed edge set into the subset losing the injection but preserving the surjective mapping.

G-Pattern Generalization ($\geq$) is a mapping between two sets of edges that is injective-only and non-surjective. Examples of *G-Pattern* generalization are the reverse operations of *G-Pattern*

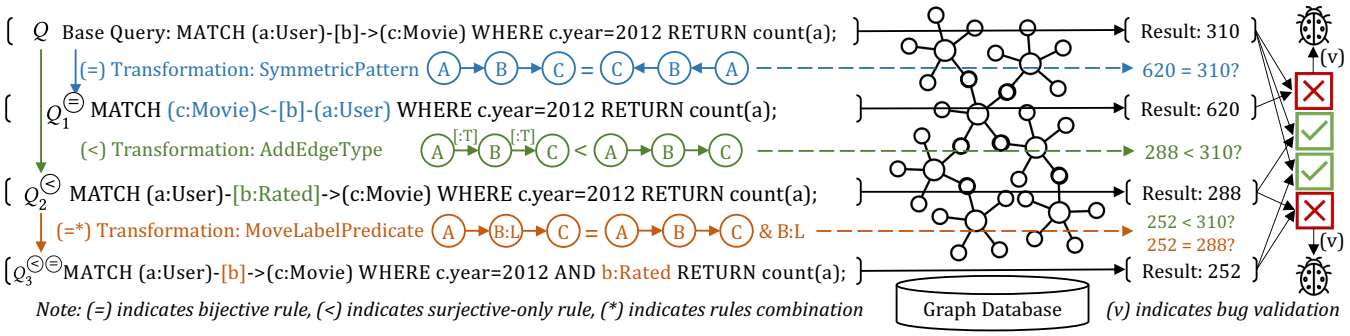

Figure 5: Overview of Injective and Surjective Graph Query Transformation

restriction (*e.g.,* expanded *G-Pattern* in Figure 4), and *G-Pattern* generalization alters the directed edge set into the superset losing the subjection but preserving the injective mapping.

4 Approach

To detect logic bugs in GDBMSs, we propose injective and surjective Graph Query Transformation (GQT), allowing mutation of graph queries in a way that respects the injective and subjective directed edge mappings between *G-Patterns*. GQT considers the following directed edge mappings, bijective mappings, and injective-only or surjective-only mappings. Based on bijective directed edge mappings, given a query Q, we generate a follow-up query Q' so that $RS(Q') = RS(Q)$ (the result sets are equal). To derive the follow-up query Q', we introduce an equivalent operator ⓐ, which randomly applies a bijective *G-Pattern* transformation. Based on injective-only or surjective-only edge mappings, given a query Q, we generate a follow-up query Q' so that $RS(Q') \supseteq RS(Q)$ or $RS(Q') \subseteq RS(Q)$, that is, the follow-up query has more or fewer results (at least equal) than the base query. To derive such comparative relations, we introduce variant operators ⓑ and ⓒ, which randomly apply an injective-only or subjective-only transformation.

Approach Overview. Our approach generates equivalent or variant mutation queries by applying query transformations to a base query and further checks the expected consistency of their results. Intuitively, our *test oracle* is that semantically equivalent queries should fetch the same results while variant queries should return with larger or smaller (at least equal) results. Figure 5 provides an overview and examples of query transformations with injective and surjective Graph Query Transformation (GQT). Given the base query Q, we utilize query transformations inspired by injective and surjective *G-Pattern* mappings to generate mutated queries (*i.e.,* $Q_1^{ⓐ}$, $Q_2^{ⓒ}$, $Q_3^{ⓒⓐ}$). Given the operators, we can infer their result relationships, thus the logic bugs can be reflected by checking the consistency among the base and mutated queries.

We list the query transformations (*i.e.,* query mutation rules) in Table 1 with ID, name, transformation type, transformation class, transformation description, and examples in *Cypher*. In terms of transformation type, we classify them into two groups: equivalent transformations that support operator ⓐ (based on bijective path mappings) and variant transformations that support operators ⓑ and ⓒ (based on injective-only or surjective-only path mappings). Due to the limited space, we have omitted the conditions for applying those transformations, such as the minimum number of edges and the presence of cycles, and only listed the one-way transformation (the reverse transformation is also applicable). As for the transformation class, we categorize graph query transformations into three groups according to their strategies: Structure-GQT (◐) rules, Property-GQT (◑) rules, and Non-GQT (○) rules. We next introduce each class of query transformations in detail.

Structure-GQT (◐) Rules. This group of rules is derived from injective and surjective path mappings we mentioned in Section 3, which alter the graph pattern while preserving the injection or surjection, or both, on the mapping between directed edge sets. One example of Structure-GQT rules is shown in Figure 5: one equivalent query $Q_1^{ⓐ}$ with the *G-Pattern* (c:movie)<-[b]-(a:user) mutated from base query Q with the *G-Pattern* (a:user)-[b]->(c:movie) by applying the transformation based on the symmetric *G-Pattern* mapping in Figure 4. Operator ⓐ indicates that its result should be equal to the base query, which is not satisfied by checking the query outputs (*i.e.,* $620 \neq 310$), revealing a logic bug. Similarly, for every injective and surjective *G-Pattern* mapping mentioned in Section 3, we formulate their query transformation (*i.e.,* SymmetricPattern, UnfoldCyclicPattern, PatternPartition, AddEdgeDirection, SpanningSubgraph, InducedSubgraph, and ExpandPattern) accordingly in Table 1 as a class of Structure-GQT rules.

Property-GQT (◑) Rules. This group of rules is derived from the labeled property graph model, the basis of modern GDBMSs we target, to enlarge the space of query transformations. Property-GQT (◑) rules take special features such as node labels, edge types, and their properties from the labeled property graph model and generate injective and surjective *G-Pattern* mappings upon them. We classify this group of rules also as injective and surjective Graph Query Transformation (GQT) due to its nature of altering *G-Patterns* in graph queries. One example of Property-GQT rule is shown in Figure 5: one query restriction $Q_2^{ⓒ}$ with the *G-Pattern* (a:user)-[b:rated]->(c:movie) mutated from the base query Q by adding the edge type (*i.e.,* [:rated]). We identify it as query restriction as the *G-Pattern* mapping is surjective-only considering the edge type, which splits the directed edge set into two disjoint subsets (*i.e.,* GP={{:rated},{:!rated}}) while the directed edge set of mutated *G-Pattern* is one of the subsets with [:rated] type (*i.e.,* GP'={:rated}). Thus, *G-Pattern* restriction (ⓒ) indicates that its result should be

Table 1: Illustrative List of Graph Query Transformations—Queries Colored with ~~DELETION~~ and ADDITION.

ID	Rule Name	Class	Type	Transformation	Example (In *Cypher*)
01	SymmetricPattern	◐	Equivalent	Replace graph pattern with a symmetric one	MATCH (A~~:MOVIE~~)--(B:MOVIE) RETURN COUNT(~~AB~~);
02	UnfoldCyclicPattern	◐	Equivalent	Unfold cyclic pattern via adding predicate	MATCH (A)--(B:MOVIE)--(C~~A~~) **WHERE A=C** RETURN COUNT(A);
03	PatternPartition	◐	Equivalent	Split graph pattern to disjoint sub-patterns	MATCH (A)-->(B:MOVIE), **(B:MOVIE)**-->(C) RETURN COUNT(A);
04	AddEdgeDirection	◐	Variant	Add edge direction to undirected edge	MATCH (A)-->(B:MOVIE) WHERE B.YEAR=2012 RETURN COUNT(A);
05	SpanningSubgraph	◐	Variant	Spanning subgraph by deleting edges	MATCH (A)-->(B:MOVIE)-->(C)~~, (A)-->(C)~~ RETURN COUNT(A);
06	InducedSubgraph	◐	Variant	Induced subgraph by deleting vertices	MATCH (A)--(B:MOVIE)--(C)~~--(D:ACTOR)~~ RETURN COUNT(A);
07	ExpandPattern	◐	Variant	Expand graph pattern by adding nodes	MATCH (A)--(B:MOVIE)--(C:MOVIE)**--(D)** RETURN COUNT(A);
08	AddNodeLabel	◑	Variant	Add node label to existing node	MATCH (A:**USER**)--(B:MOVIE) WHERE NOT A=B RETURN COUNT(A);
09	AddEdgeType	◑	Variant	Add edge type to existing edge	MATCH (A:USER)-**[R:RATED]**-(B:MOVIE) RETURN COUNT(A);
10	MoveLabelPredicate	◑	Equivalent	Move node label to the predicate	MATCH (A~~:USER~~)--(B:MOVIE) **WHERE A:USER** RETURN COUNT(A);
11	CountIdProperty	◑	Equivalent	Count the node id property	MATCH (A:USER)--(B:MOVIE)-->(C) RETURN COUNT(**ID(A)**);
12	CountOtherName	◑	Equivalent	Count other name in the same path	MATCH (A:USER)--(B:MOVIE)-->(C) RETURN COUNT(~~AC~~);
13	DisjointPredicate	○	Equivalent	Split predicate into disjoint parts	MATCH (A) WHERE A.P>0 ~~AND~~**WITH * WHERE** A.Q>0 COUNT(A);
14	RedundantPredicate	○	Equivalent	Append alway-true condition to predicate	MATCH (A:USER)–(B:MOVIE) **WHERE NOT A=B** RETURN COUNT(A);
15	RenameVariables	○	Equivalent	Rename node or edge variables	MATCH (~~AN~~)--(~~BM~~:MOVIE) WHERE ~~AN~~:USER RETURN COUNT(~~AN~~);
16	AddCallWrapper	○	Equivalent	Return results by calling the function	**CALL {** MATCH (A:USER) RETURN COUNT(A) **AS X } RETURN X**;

less equal than the base query, which is satisfied to pass the test oracle (*i.e.,* 288 ≤ 310). We generate five Property-GQT rules in total (*i.e.,* AddNodeLabel, AddEdgeType, MoveLabelPredicate, CountIdProperty, CountOtherName) listed in Table 1 as the complement of *G-Pattern* mappings on the labeled property graph model.

Non-GQT (○) Rules. This group of rules is derived from query syntax or other query elements except *G-Pattern*. Specifically, Non-GQT rules do not mutate the *G-Pattern* explicitly and does not use *G-Pattern* mapping. As shown in Table 1, we create four Non-GQT rules, namely, DisjointPredicate splits the *Predicate* into two parts and applies them accordingly; RedundantPredicate appends an always-true condition (*e.g.,* WHERE ID(A)>=0, ID() returns the identifier) to the predicate; RenameVariables changes the variable names; AddCallWrapper aims to execute the graph query in a calling function way. We involve this group of rules in our approach to facilitate generating diverse mutated queries by rules combination.

Rules Combination. GraphGenie supports rules combination and iteration to generate more diverse testing queries. The equivalent operator ⊜ can be combined with one variant operator (*i.e.,* ⊃ or ⊂), and all operators can also be iteratively applied. For example, in Figure 5, another query $Q_3^{⊂⊜}$ with the *G-Pattern* (A:USER)-[B]-(C:MOVIE) and additional predicate B:RATED is generated via rules combination by using MoveLabelPredicate bijective query transformation on $Q_2^{⊂}$. Given the bijective query transformation, we identify the query equivalence and notice the unexpected inconsistency (*i.e.,* 252 ≠ 288) between $Q_2^{⊂}$ and $Q_3^{⊂⊜}$, revealing a logic bug. We believe rules combination and recursiveness can facilitate generating a wider range of mutated queries.

Correctness Validation. We compare the results of the base query Q and mutated query Q' in terms of operators (*i.e.,* ⊜, ⊃, and ⊂). When performing equivalent transformations, we can derive the operator as ⊜. For variant transformations, determining the appropriate operator (⊃ or ⊂) requires the *G-Pattern* mappings and the context of the query. For instance, consider two pairs of query transformations using InducedSubgraph, as listed in Listing 2. Despite both using the same query transformation, they may have different operators depending on the semantics of the *Return* clause.

$$Q' = \begin{cases} Q^{⊜}, & \text{if } \{⊜\} \in Op \ \wedge \ \{⊃, ⊂\} \notin Op \\ Q^{⊃}, & \text{if } \{⊃\} \in Op \ \wedge \ \{⊂\} \notin Op \\ Q^{⊂}. & \text{if } \{⊂\} \in Op \ \wedge \ \{⊃\} \notin Op \end{cases} \quad (1)$$

Given the set of applied operators (*i.e., Op*), as shown in Equation 1, we define the mutated query as an equivalent query when only applying the equivalent operator; similarly, a generalized variant query $Q^{⊃}$ when only applying operator ⊜ and ⊃; a restricted variant query $Q^{⊂}$ when only applying operator ⊜ and ⊂.

$$RS(Q^{⊜}) = RS(Q),\ RS(Q^{⊃}) \supseteq RS(Q),\ RS(Q^{⊂}) \subseteq RS(Q) \quad (2)$$

As shown in Equation 2, the equivalent mutated query $Q^{⊜}$ should share the same *result set* (*i.e., RS*()) with its base queries Q while variant mutated query $Q^{⊃}$ or $Q^{⊂}$ should return more or fewer results (at least equal). If any inconsistency exists, we further verify the suspicious queries that give inconsistent results by checking whether they are reproducible.

Generalizability. Graph queries exhibit common patterns such as nodes, edges, and clauses, which persist across different graph query languages, allowing for the possibility of general graph query transformations. Our study has identified several common graph-matching query elements in graph query languages, including *Match*, *G-Pattern*, *Predicate*, *Return*, and *Others*, which are summarized in Section 2. Although graph query languages represent graph patterns using different syntax, they consistently use vertices and edges, making it feasible to apply universal query transformations such as SymmetricPattern and PatternPartition. Our query transformations are based on common graph query elements, with the goal of being compatible with most GDBMSs based on the labeled property graph model.

Listing 2: Different Variant Operators with Same Patterns

```
Q1: MATCH (a)-[]-(b)-[]-(c) RETURN count(a);
Q1⊂: MATCH (a)-[]-(b) RETURN count(a);
Q2: MATCH (a)-[]-(b)-[]-(c) RETURN count(DISTINCT a);
Q2⊃: MATCH (a)-[]-(b) RETURN count(DISTINCT a);
```

5 Implementation

In this section, we introduce the implementation details of GraphGenie, which is implemented in Python with over 2K lines of code.

GraphGenie includes four main components, namely, schema scanner, base query generator, GQT query mutator, and bug validator. The schema scanner acquires information about the testing schema, including labels, properties, and node connectivities. This information is then used by the base query generator with correct syntax to construct valid base queries. The GQT query mutator generates mutated queries by applying graph query transformations to each base query, either through a single rule or random combinations. GraphGenie then examines the results of mutated queries (*i.e.,* $Q^{\textcircled{=}}$, $Q^{\textcircled{>}}$, $Q^{\textcircled{<}}$) in the target GDBMS to identify potential bugs. Finally, the bug validator evaluates each potential bug, reduces faulty queries, and generates bug reports.

Schema Scanner. In GraphGenie, we collect schema information at the beginning of GDBMS testing to assist in creating meaningful queries. Specifically, GraphGenie executes a series of predefined queries (*e.g.,* using LABELS(), TYPE(), KEYS() functions) to fetch metadata of the target schema including labels, properties. Such information helps GraphGenie generate base queries with valid values. In addition, GraphGenie leverages connectivity among nodes to avoid generating classes of queries that are guaranteed to yield empty results. For instance, an (:USER) node could never connect to a (:GENRE) node in the Recommendation[4] dataset, thus GraphGenie avoids creating graph patterns like (:USER)-->(:GENRE) by referring to the connectivity matrix between nodes.

Base Query Generator. Generated queries are always syntactically correct and aware of the schema, aiming to exercise deeper logic of the GDBMSs. To guarantee syntactic correctness, we followed the official grammar of graph query languages as other works (*e.g.,* GDSmith [27] and Grand [55]) did. For *schema awareness*, using information from the schema scanner, GraphGenie generates base queries with valid labels and properties to prevent retrieving non-existing graph patterns, also avoiding parts of empty-results queries by checking node connectivity.

While base queries generated by GraphGenie are effective in identifying logic bugs in GDBMSs, there are many existing approaches that aim to generate base queries with higher efficiency and coverage like SQLancer [44] and SQLsmith [1]. Those orthogonal works can be combined with GraphGenie in a GDBMS context to further improve the testing efficiency.

GQT Query Mutator. GraphGenie uses GQT to create mutated queries. In theory, GraphGenie has the ability to generate unlimited mutated queries with test oracles via rules combination. However, we usually limit the number of mutated queries through a configurable parameter of GraphGenie. We also avoid rules combination of operators $\textcircled{>}$ and $\textcircled{<}$ as the result would be uncertain.

Bug Validator. The bug validator checks the result of the base query and mutated query. In terms of three operators, any discrepancy against Equation 2 indicates a potential logic bug. GraphGenie validates the potential bug by checking whether it is reproducible.

[4] https://github.com/neo4j-graph-examples/recommendations

Table 2: Information of GraphGenie Target GDBMSs

GDBMS	Rank	Github Stars	Init Release	LoC
Neo4j	1	11.2k	2007	1,241K
RedisGraph	3*	1.8k	2018	1,618K
AgensGraph	37	1.3k	2017	2,035K
TinkerPop	26	1.7k	2009	459K
JanusGraph	7	4.8k	2017	172K
HugeGraph	22	2.2k	2018	146K

Query Language Support. We implemented GraphGenie to support two popular graph query languages (*Cypher* and *Gremlin*). GraphGenie supports *Cypher*, which is the industry's most widely used property graph query language. For *Gremlin*, we adopt an official Gremlin server plugin called *Cypher for Gremlin*[5] published by OpenCypher.[6] It is developed for users of Apache TinkerPop [4] to allow querying *Gremlin* databases with *Cypher*. The translation process converts *Cypher* query to an internal representation, which is subsequently parsed by a set of rewrites to output *Gremlin* representations. According to the official coverage testing report [36], over 98% Gremlin steps and over 91% of the scenarios have been supported. We collect the translated queries to test the incompatible latest versions or other *Gremlin* GDBMSs that are not officially supported by this plugin.

6 Evaluation

In this section, we answer the following research questions to assess various important aspects of GraphGenie:

- **Q1 Discovery of Unknown Bugs**. How effective is GraphGenie in discovering unknown bugs in mature GDBMSs?
- **Q2 Comparison with Existing Techniques**. How does GraphGenie's effectiveness on GDBMS testing compare with the state-of-the-art approaches, such as Grand [55] and GDBMeter [22]?
- **Q3 GQT Contribution Analysis**. To what extent do the individual rule categories contribute to GraphGenie's performance?
- **Q4 Applicability of Detecting Performance Issues**. Is GraphGenie also applicable to finding performance issues in GDBMSs?

Target GDBMSs. We spent most of our time testing *Cypher* GDBMSs so we focus on them in the evaluation. We first selected three top-ranking *Cypher* GDBMSs to evaluate GraphGenie's effectiveness comprising Neo4j [37], RedisGraph [39], and AgensGraph [7]. Table 2 shows their DB-Engine Ranking [15] of GDBMSs, GitHub stars, initial release date, and Lines of Code (LoC)[7]. We also applied our approach in three *Gremlin* GDBMSs including TinkerPop [4], JanusGraph [19], and HugeGraph [18] for comparison with existing works to demonstrate the approach's generality to other graph query languages.

Table 3: Unknown Logic/Error Bugs Found by GraphGenie

	Logic Bugs			Internal Errors	
GDBMS	**Unconfirmed**	**Confirmed**	**Fixed**	**Fixed**	**Total**
Neo4j	0	0	2	3	5
RedisGraph	1	3	1	0	5
AgensGraph	0	0	3	0	3
Gremlin-DBs	6	0	0	0	6
Total	7	3	6	3	19

[5] https://github.com/opencypher/cypher-for-gremlin
[6] http://opencypher.org/
[7] Statistics on March 2023. RedisGraph rank includes the secondary database models.

Listing 3: RedisGraph Logic Bug via Counting Property

```
Q: MATCH ()-[a]-() RETURN count((a));
   // Result: 20000    Response Time: 0.20ms
Q^(=): MATCH ()-[a]-() RETURN count(id(a));
   // Result: 39993    Response Time: 55.57ms
```

Listing 4: RedisGraph Logic Bug via Symmetric Path

```
Q: MATCH (a:A)-[*1..2]-(b:B) return count(1);
   // Result: 204    Response Time: 0.76ms
Q^(=): MATCH (b:B)-[*1..2]-(a:A) return count(1);
   // Result: 238    Response Time: 0.75ms
```

Listing 5: Neo4j Internal Error via Mutated Variable Name

```
Q: OPTIONAL MATCH p=(n)-->()--()-->()--()--()-->()<--(n)
   RETURN id(n) SKIP 2 LIMIT 1;  // OK
Q^(=): OPTIONAL MATCH p=(a)-->()--()-->()--()--()-->()<--(a)
   RETURN id(a) SKIP 2 LIMIT 1;  // Exception
```

Listing 6: Neo4j Logic Bug via Moving Edge Type

```
Q: OPTIONAL MATCH (s1)-[s0:DIRECTED]-()<-[s2:ACTED_IN]-(s1
   ) RETURN count(s1);  // Result: 43
Q^(=): OPTIONAL MATCH (s1)-[s0]-()<-[s2:ACTED_IN]-(s1) WHERE
   s0:DIRECTED RETURN count(s1);  // Result: 491
```

Testing Datasets. We used two standard datasets created by Neo4j, that is, CyberSecurity[8] and Recommendation[9]. CyberSecurity includes data from active directory environment auditing and analysis of possible attack paths using graphs, which includes 953 nodes and 4,858 relationships. Recommendation is the dataset of movie reviews, which has 28,863 nodes and 166,261 relationships.

Experiment Infrastructure. We performed all experiments on a personal computer with Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz, 16GB RAM. The OS is Ubuntu 20.04.2 LTS.

6.1 Discovering Unknown Bugs

To detect new logic bugs in GDBMSs, we intermittently tested the latest versions of the target GDBMSs over a period of three months, which is a typical methodology for evaluating the effectiveness of automatic testing tools [22, 44]. We reported bugs after reducing bug-inducing queries and checking whether the issue had already been reported on issue trackers to avoid duplicate bug reports.

Results. Table 3 provides a summary of the number of previously unknown logic bugs and internal errors—we discuss performance issues that we identified in Section 6.4. We classified the unknown bugs into three disjoint categories:

- *Unconfirmed* bugs refer to the bugs that have been identified and submitted but are awaiting further investigation by developers to confirm the root cause.
- *Confirmed* bugs refer to the bugs that have been acknowledged for their existence by developers but have not been fixed.
- *Fixed* bugs refer to the bugs that have been confirmed and patched by the developers.

In total, we identified 19 unknown bugs[10], of which 3 were confirmed and 9 were fixed. Next, we present several noteworthy bugs that GraphGenie identified, categorizing them based on their root cause through our analysis of developers' feedback or fix commits. We classified them into several different classes.

Incorrect Relationship Counting. Edges are essential to connect nodes in graph queries. Modern GDBMSs support multiple advanced features in the relationships/edges such as variable path lengths, making it complex to count relationships when sophisticated graph patterns are used.

Bug 1: Listing 3 shows one relationship counting bug[11] detected by GraphGenie using the CountIdProperty query transformation. The RETURN clause using the count aggregation is mutated to count the universal node properties (*e.g.,* ID() returns the identifier), which is expected to output the same results regarding the equivalent transformation. We found this bug in RedisGraph, because the base query returned 20,000, and the follow-up query returned 39,993. The developers quickly acknowledged this bug and explained that RedisGraph COUNT() would miss relationships if not referenced elsewhere in the query.

Bug 2: We found another incorrect relationship counting bug[12] shown in Listing 4 by executing a pair of SymmetricPattern equivalent queries. The symmetric query pair is expected to output the same result—retrieving the graph pattern whose endpoints are (A) and (B). The different results demonstrate a logic bug. After analyzing the fix commit by the RedisGraph developers, we confirm the root cause of this bug was the incorrect logic to stop expanding a path upon detecting a cycle.

Unrobust Query Plan Operator - Expand(Into). In Neo4j [37], *Expand(Into)* is one important operator, which exists in various query plans. When both the start and end nodes have already been found, *Expand(Into)* operator is used to find all relationships connecting the two nodes. We found two bugs related to this operator.

Bug 3: Listing 5 illustrates an unknown bug[13] related to *Expand(Into)*. GraphGenie applied the RenameVariables query transformation to mutate the node variables in a doubly-connected loop. With different variable names, the query execution aborted unexpectedly and further caused one server-side exception. Neo4j developers have fixed this bug. By analyzing the fix commit, we found the root cause is that *Expand(Into)* did not handle doubly-connected loops combined with SKIP and LIMIT operations.

Bug 4: Another related logic bug[14] shown in Listing 6 is detected with the query transformation MoveLabelPredicate. We moved the edge type [S0:DIRECTED] from the *G-Pattern* to the query predicate. This transformation caused Neo4j to fail to find paths that satisfy the predicate, revealing a logic bug.

[8] https://github.com/neo4j-graph-examples/cybersecurity
[9] https://github.com/neo4j-graph-examples/recommendations
[10] *Gremlin* queries are generated through the Cypher for Gremlin plugin. We discovered that the plugin may lead to buggy translations, so some Gremlin bugs have been categorized as *Unconfirmed* bugs.
[11] https://github.com/RedisGraph/RedisGraph/issues/2744
[12] https://github.com/RedisGraph/RedisGraph/issues/2865
[13] https://github.com/neo4j/neo4j/issues/12968
[14] https://github.com/neo4j/neo4j/issues/12991

Listing 7: AgensGraph Logic Bug via Pattern Partition

```
Q: MATCH (a)-[*1..1]->(b) RETURN count(a);
     // Result: 100
Q^∈: MATCH (a),(a)-[*1..1]->(b) RETURN count(a);
     // ERROR:  column "a" does not exist
```

Unrobust Variable-Length Expression Parsing. Variable-length expressions (VLEs) are one common feature that allows setting variable lengths between nodes for fuzzy matching in GDBMSs.

Bug 5: As shown in Listing 7, GraphGenie detects the logic bug[15] in AgensGraph [7] by PatternPartition, which split the graph pattern into two parts. As the split pattern does not yield additional constraints for the query, they are considered equivalent queries. However, parsing the VLE in the $Q^{\in}$ makes the node (A) invisible to the aggregation function, which unexpectedly caused the error.

6.2 Comparison with Existing Techniques

We sought to compare GraphGenie with state-of-the-art GDBMS testing techniques: Grand [55] and GDBMeter [22] to investigate how GraphGenie compares with current techniques. We analyzed and compared the effectiveness of finding confirmed bugs among them. We could not compare with GDSmith [27], a similar testing technique to Grand, as its tool is not publicly available.

Comparison with Grand. Grand [55] is the state-of-the-art approach that uses differential testing to detect logic bugs in *Gremlin* GDBMSs. The core idea behind Grand is to execute the same queries on multiple GDBMS instances and check the consistency of their results. We observed a significant limitation of Grand is that it produces false alarms due to its assumption—different GDBMSs have consistent semantic implementation towards the same syntax. We found that this assumption does not always hold in practice. For instance, a basic edge counting query MATCH ()-[A]-() RETURN COUNT(A) could differ between Neo4j and Memgraph [34] because of different and intended semantics in counting undirected edges, that is, by matching ()-[N]-() Memgraph gets the same edge twice (two directions) while Neo4j deduplicates.

We followed the instructions provided in the Grand GitHub repository and executed Grand to apply differential testing to JanusGraph [19], TinkerGraph [4], and HugeGraph [18]. We ran Grand for five iterations, generating a total of 500 queries. We randomly sampled 30 bugs from these queries and found that over 80% of them were false alarms. In contrast, when using GraphGenie, we were able to identify the same number of potential logic bugs and our test oracle only identified bugs caused by semantic discrepancies, ensuring that GraphGenie only reports true logic bugs.

To further investigate the false alarms, we categorized Grand's results according to the root cause below: Many false alarms (56.66%) were caused by unsupported syntax. For instance, for the INSIDE() function, HugeGraph lacks support for the BOOLEAN data type in this function, resulting in an exception when using INSIDE(FALSE, TRUE), which caused the difference compared to other GDBMSs. In 10% of the cases, false alarms were caused by differences in the exception types across different GDBMSs (*e.g.,* when parsing LTE(-1), *IllegalArgumentException* in HugeGraph and *ExecutionException* in JanusGraph). For another 10% of the cases, false alarms were caused by differences in the encoding across different GDBMSs. For instance, when parsing the query G.E().VALUES('EP2'), three GDBMSs instances gave the same output but GDBMeter classified it as bug-inducing due to Unicode characters being represented differently in various encodings. Some false alarms (6.66%) can occur when different GDBMSs have inconsistent semantics in graph queries (*e.g.,* G.V().HAS('VP3',0.93463105)) may produce different results in TinkerPop). In the remaining cases, we were unable to assess whether these test cases were false alarms or true bugs. They would need further examination by developers.

Listing 8: Example of GDBMeter Missed Bug

```
Q: MATCH (a)-[]-(a) RETURN count(a);
     // Base Query Result: 200
MATCH (a)-[]-(a) WHERE id(a)>=1.0 RETURN count(a);
     // (TLP-True) Result: 200
MATCH (a)-[]-(a) WHERE NOT id(a)>=1.0 RETURN count(a);
     // (TLP-False) Result: 0
MATCH (a)-[]-(a) WHERE id(a)>=1.0 IS NULL RETURN count(a);
     // (TLP-Null) Result: 0
Q^∈: MATCH (a)-[]-(b) WHERE a=b RETURN count(a);
     // (GraphGenie) Result: 16
```

We discovered that, besides high false alarms, differential testing tends to have missed bugs when the same incorrect answers are returned from different GDBMSs, particularly those that utilize the *Gremlin* language. This is mainly because many Gremlin GDBMSs share a codebase inherited from Apache TinkerPop [4], which can lead to common logic bugs in the different GDBMSs. For instance, the logic bug (*Unrobust Alias Parsing*) shown in Listing 7 was present in all the GDBMSs tested by Grand, causing a differential-testing approach to overlook such kinds of bugs.

Comparison with GDBMeter. GDBMeter[16] [22] is a recently-proposed metamorphic testing technique for detecting bugs in GDBMSs that uses the approach called *Ternary Logic Partitioning* [42]. Ternary Logic Partitioning was first introduced for testing relational database systems and based on the insight that a boolean predicate p can yield one of three possible outcomes: *True*, *False*, or *Null*. A query can be divided into three distinct sub-queries that operate on rows or intermediate results based on the conditions of p, *NOT* p, and p *IS NULL*, respectively. A bug is detected when the combined results of the three sub-queries do not match the result of the original query.

Ternary Logic Partitioning (TLP) is known as a general technique for detecting logic bugs in database systems. It covers only partial graph query mutations on predicates that occur in GDBMSs. To evaluate the effectiveness of our approach compared to GDBMeter in detecting logic bugs, we used the same methodology as prior works [41, 42], that is, to conduct a manual and best-effort analysis to (1) identify any bugs found by GraphGenie that were overlooked by GDBMeter, and (2) determine whether GraphGenie can detect confirmed bugs already detected by GDBMeter. In addition, to further validate our findings, we conducted a best-effort analysis of the bug-inducing test cases generated by them over 24 hours.

[15] https://github.com/bitnine-oss/agensgraph/issues/609

[16] We contacted the author, who kindly provided the preprint and access to the tool

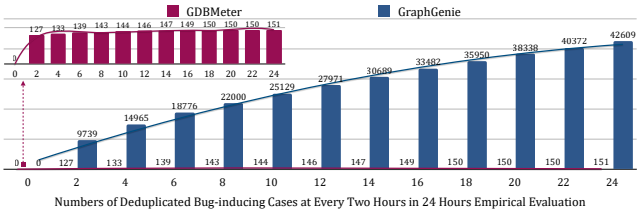

Figure 6: The 24 Hours Empirical Testing Statistics.

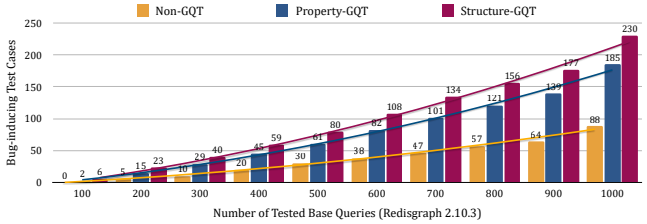

Figure 7: GQT Contribution Analysis

Bugs Missed by GDBMeter. We manually analyzed all confirmed and fixed logic bugs detected by GraphGenie and tested whether they can be detected via Predicate Partitioning. For implementing Ternary Logic Partition, we split each base query of the confirmed bug via adding an identity-comparison (*e.g.,* `WHERE ID(A)>0`) predicate supported by most GDBMSs. We inspected the results of the base query and its ternary queries to check whether it split into disjoint subsets. Out of 9 bugs that are applicable to Ternary Logic Partition, GDBMeter was able to detect only 3 bugs. Listing 8 demonstrates one GDBMeter missed bug[17] that GraphGenie found by utilizing the UnfoldCyclicPattern rule to mutate the base query. The results showed that Ternary Logic Partition does not necessarily reveal inconsistency among *True*, *False*, and *Null* queries.

Detecting Bugs found by GDBMeter using GQT. We also tried to detect bugs found by GDBMeter using our approach. Out of 36 confirmed or fixed bugs found by GDBMeter in GDBMSs, we notice that 29 of them are internal errors (*i.e.,* exceptions in Java or crashes in C/C++) that rely on query generation rather than the test oracle, hence those bugs are orthogonal to our approach. We can detect all remaining 7 logic bugs found by GDBMeter using GQT. For two logic bugs fixed in Neo4j, we next demonstrate how GraphGenie detects them via graph query transformations.

Considering the logic bug[18] found by GDBMeter, GraphGenie mutated the base query `MATCH (N:L) WHERE N.P STARTS WITH LTRIM(N.P) RETURN COUNT(N)` by using the MoveLabelPredicate transformation (*i.e.,* delete `:L` and add `WHERE "L" IN LABELS(N)`). The mutated query was expected to return the same result. However, the count increased from 0 to 1 unexpectedly, indicating this logic bug could also be revealed by our tool. Similarly, GraphGenie detected another logic bug[19] found by GDBMeter using the same strategy.

GDBMeter has been used to find logic bugs in GDBMSs. However, our results show that it is limited to testing the correct handling of predicates and does not deal with finding logic bugs in the handling of graph patterns. We believe GraphGenie could be a good complement in this direction and better improve the effectiveness.

The 24 Hours Empirical Testing. Although it is difficult to have a fair and direct comparison between testing techniques, we conducted a best-effort empirical testing comparison between GDBMeter and GraphGenie to illustrate their differences. We ran GDBMeter and GraphGenie on RedisGraph (v2.11.3, the latest version at this time) supported by both tools for 24 hours, which is the typical and suggested time budget for fuzzing techniques [23]. Towards a fair comparison, both tools are running on the same dataset randomly generated by GDBMeter. Over a 24-hour testing period, GDBMeter discovered 1,053,128 bug-inducing cases and GraphGenie discovered 884,279 bug-inducing cases. To estimate how many of the bug-inducing test cases indicate unique bugs, we applied a best-effort approach to duplicate the bug-inducing cases by retaining only one pair for each result count pair as a coarse approximation. For both systems, they compare one base query result n with a mutated result m where m stands for the sum of count results of TLP queries in GDBMeter and the count result of the mutated query in GraphGenie. Given two bug-inducing test cases with results (n_1, m_1) and (n_2, m_2), they yield duplicated count results pairs when n_1=n_2 and m_1=m_2. We obtained 151 deduplicated bug-inducing cases for GDBMeter and 42,609 for GraphGenie. We believe that these numbers likely overestimate the number of found bugs, but they give insights into the relative bug-finding capabilities. Figure 6 shows that GraphGenie found two orders of magnitude more unique bug-inducing cases within 24 hours and kept the increasing trend after 24 hours when GDBMeter seemed to reach a peak due to limited graph patterns.

6.3 GQT Contribution Analysis

We investigated the contribution of individual rule categories to our approach's overall effectiveness. As shown in Table 1, our approach includes three groups of graph query transformations: Structure-GQT (◐), Property-GQT (◑), Non-GQT (○). We analyzed to what extent each group of query transformation contributes to the improvement of GraphGenie on GDBMS testing by comparing the number of found bug-inducing cases. In order to conduct a fair comparison, we chose three representative graph query equivalent transformations for each group (*i.e.,* 01,02,03 for ◐; 10,11,12 for ◑; 13,14,15 for ○, as shown in Table 1), and disabled rules combination. We executed 1000 base queries with 6914 mutated queries in RedisGraph 2.10.3, and recorded the results of bug-inducing test cases after every hundred executions of the base queries.

The results of our contribution analysis are presented in Figure 7. We demonstrate three groups of graph query transformations in an accumulative way to show our improvement (*i.e.,* Non-GQT results only count Non-GQT rules; Property-GQT results count both Non-GQT rules and Property-GQT rules; Structure-GQT results count all three groups of rules). The statistics show that non-GQT rules are effective in finding bug-inducing test cases while using GQT rules facilitates uncovering more bug-inducing cases in testing GDBMS. When not considering graph pattern mutations, Non-GQT rules were only able to identify around 40% of bug-inducing test cases compared to GraphGenie. These findings provide evidence that our insight into mutating the *G-Pattern* in queries is effective in generating more diverse mutated queries that are more likely to uncover logic bugs.

[17] https://github.com/bitnine-oss/agensgraph/issues/595
[18] https://github.com/neo4j/neo4j/issues/12887
[19] https://github.com/neo4j/neo4j/issues/12884

Listing 9: Performance issue GraphGenie Found in Neo4j

```
Q: MATCH p=(n1)--()<-[r1]-()--(r2:PostCode)--(n2)<--()
    -->()-->()<-[r3:POSTCODE_IN_AREA]-(n1) RETURN n2 as X
     SKIP 6 LIMIT 2;     // Response Time: 4642011ms
Q^[⊵]: CALL { {Q} } RETURN X; // Response Time: 5984ms
```

Listing 10: Performance issue GraphGenie Found in Neo4j

```
Q: MATCH (s3:Computer)<--(s4:HighValue)-->(s0:Computer)
    -->(s2:User)--()-->(s1:User)<--() RETURN s3 ORDER BY
    id(s3) ASC SKIP 4 LIMIT 6; // Response Time: 100ms
Q^[=]: OPTIONAL {Q}; // Response Time: 201384ms
```

6.4 Finding Performance Issues

Performance issues are another type of important bug in GDBMSs that negatively affect query response times. One key challenge in detecting performance issues in GDBMSs is establishing a reliable test oracle that can accurately specify the expected behavior (*i.e.,* response time) of performant GDBMSs for a particular SQL query. While we primarily sought to find logic bugs in GDBMSs, we believe that our key approach GQT would, in principle, also be applicable to finding performance issues.

To detect performance issues in GDBMSs, our approach reuses graph query transformations in Table 1, but adopts new operators to mutate the base query for a similar ($[=]$) or variant ($[\trianglerighteq]$ or $[\trianglelefteq]$) execution time. Similar to detecting logic bugs, given a base query Q, our performance test oracle is to check whether a mutated query Q' has expected execution time in terms of operators.

We use a configurable threshold T to distinguish the similar execution time or the unexpected deviation. Similar to existing testing works [21, 30], we set the empirically-determined threshold as $T^{[=]} = 5\times$ considering operation $[=]$ in our experiments to balance false alarms and missed detections and it is more conservative compared to 2× used in AMOBEA [30]. We follow Equation 1 to define mutated queries with new operators to be equivalence, restriction, or generalization in detecting performance issues. Equation 3 illustrates that the base query Q and equivalent mutated query $Q^{[=]}$ should not have a performance deviation that exceeds a factor of 5× in terms of query execution times. For variant operators, we set the threshold as $T^{[\neq]} = 2\times$. Equation 4 illustrates that the variant mutated query $Q^{[\trianglerighteq]}$ should not be less than the half of query execution time of base query Q. while Equation 5 shows $Q^{[\trianglelefteq]}$ should not exceed 2× query execution time of base query Q.

$$max(time(Q), time(Q^{[=]})) \leq min(time(Q), time(Q^{[=]})) \times T^{[=]} \quad (3)$$

$$time(Q^{[\trianglerighteq]}) \times T^{[\neq]} \geq time(Q) \quad (4)$$

$$time(Q^{[\trianglelefteq]}) \leq time(Q) \times T^{[\neq]} \quad (5)$$

We tested the capability of detecting performance issues for one GDBMS—Neo4j. Out of 10 submitted issues, we received positive feedback for 6 performance issues shown in Table 4 and the difference in response times is large, varying from 10× to 10^3×. We found the results were encouraging as diagnosing and fixing performance issues is often more challenging than logic bugs.

Table 4: Performance Issues Found by GraphGenie in Neo4j

Bug ID	Status	Time(Q)	Time(Q')	Developer Feedback
12973	Fixed	4642011ms	5984ms	A fix will come with the next release
13034	Fixed	100ms	201384ms	A fix will come with the next release
13010	Confirmed	77ms	12147ms	Bad plan but low priority to optimize
12957	Confirmed	13933ms	22ms	A suboptimal plan in old version
13003	Intended	165547ms	332ms	Query plan is suboptimal but intended
13033	Intended	1402ms	16585ms	Inaccurate estimated rows and bad plan

We next demonstrate two fixed performance issues that both incur significant slowdown in Neo4j. The first performance issue[20] we show incurs 776× slowdowns in Neo4j. As shown in the Listing 9, the mutated query uses the calling function way of executing the base query, which is expected to have a similar or higher execution time compared to the base query. However, the results showed that $Q^{[\trianglerighteq]}$ finished much faster than base query Q, revealing a less-common way of executing a Cypher query resulted in better performance. This bug has been acknowledged as an issue in SKIP clause and has been fixed in the latest Neo4j release.

Listing 10 shows a performance issue[21] resulting in a suboptimal query plan is used in OPTIONAL clause that leads to unreasonable response times. After investigation, the developer replied to the issue stating that Neo4j picked a worse query plan due to an incorrect assumption when using ORDER BY combined with OPTIONAL.

GraphGenie may have false alarms when finding performance issues in GDBMSs since query optimizers involve various tradeoffs and not every query is expected to be optimized well.

7 Related Work

We briefly summarize the most relevant related work.

Testing GDBMSs. GDBMSs have been widely adopted, so their reliability has attracted increasing attention. We briefly summarize the state of the art, which we also already discussed throughout the paper. GDsmith [27] leverages differential testing to find logic bugs in *Cypher*-based GDBMSs. Similarly, Grand [55] leverages differential testing to find logic bugs in *Gremlin*-based GDBMSs. GDBMeter [22] leverages *Ternary Logic Partitioning (TLP)* [43] to find discrepancies in handling predicate operations. Similar to TLP, GQT is a metamorphic testing approach; however, in contrast to TLP, we designed the rules to be specific to GDBMSs.

Detecting memory errors in DBMSs. Most previous methods for testing DBMSs have focused on memory errors, finding which does not require an explicit test oracle. Grey-box fuzzers, such as AFL [49], use code coverage as guidance to mutate test cases for detecting memory errors. These mutation-based fuzzers have detected many bugs in widely-used software, such as *libpng* and *OpenSSL*. However, for DBMSs, the mutation methods typically incur invalid test cases due to the highly-structured SQL grammar. Squirrel [56] uses a syntax-preserving mutation method to increase the rate of valid test cases during mutation. Generation-based methods, such as SQLSmith [1], DynSQL [20], and ADUSA [30], generate test cases according to grammar. Griffin [17] uses a grammar-free test case generation method to alleviate the human effort to construct grammar for each tested DBMS. While these works have proposed efficient ways to generate test cases, they have not tackled the test oracle problem to find logic bugs.

[20] https://github.com/neo4j/neo4j/issues/12973
[21] https://github.com/neo4j/neo4j/issues/13034

Detecting logic bugs in DBMSs. Logic bugs, which refer to incorrect results returned by DBMSs, are difficult to detect as they require a *test oracle*, a mechanism that decides whether the test case's result is expected. Song et al. proposed the oracle, Differential Query Execution (DQE) [46], to detect logic bugs in GDBMS by checking whether SQL queries with the same predicate access the same rows. Rigger et al. proposed the oracles PQS [44], NoREC [41], and TLP [43] to detect logic bugs in relational DBMSs by checking results' consistency of several related queries, and have found hundreds of bugs. To generate test cases for such test oracles, SQLRight [26] leverages code coverage to guide the test case generation, and QPG [5] guides the test cases toward unseen query plans aiming to generate the test cases that trigger diverse behaviors. While these approaches focus on relational DBMSs, we propose a novel mutation strategy injective and surjective Graph Query Transformation (GQT) in the context of testing GDBMS. In this work, we focus on the test case generation problem and propose a novel mutation strategy injective and surjective Graph Query Transformation (GQT) to detect logic bugs for GDBMSs.

Detecting performance issues. Performance issues may result in suboptimal performance, to which DBMSs are sensitive. Existing benchmarking suites, such as TPC-H [48], expose performance issues by evaluating the DBMS' performance on a set of benchmarks. APOLLO [21] compares the execution times of a query on two versions of a DBMS to find performance regression issues. AMOEBA [30] compares the execution time of an equivalent pair of queries to identify an unexpected slowdown. While GraphGenie primarily detects logic bugs, our evaluation has demonstrated that it can be used to find performance issues as well via GQT.

Metamorphic testing. One of the most successful methods to detect logic bugs is metamorphic testing [11, 13], which generates two semantic-comparable test cases and validates the consistency of results. Metamorphic testing uses an input I to a system and its output O to derive a new input I' (and output O'), for which a test oracle can be provided that checks whether a so-called *Metamorphic Relation* holds between O and O'. Conceptually, GraphGenie also belongs to metamorphic testing. Given a query I and execution result O, GraphGenie derives another query I' via GQT and checks the consistency of O and I''s result O'. Metamorphic testing has been applied successfully in various domains, including bioinformatics [12, 38], web services [9], embedded systems [24], datalog engines [31] and compilers [25]. In this work, we propose a novel metamorphic testing method to test GDBMSs.

Differential testing. Another line of research to detect logic bugs is differential testing [33], which detects bugs by executing a test case using multiple versions or instances of systems that implement the same semantics, and any discrepancy indicates a potential bug in one of these systems. Researchers have utilized this method to detect bugs across various domains, such as web services [10], Java Virtual Machine (JVM) implementations [14], compilers [53], and network protocols [8]. Differential testing was applied to testing DBMSs as a system called RAGS [45], which executes a query on multiple different DBMSs and compares their results. APOLLO [21] also applies differential testing to detect performance issues by comparing the execution time of the same query on different versions of the same DBMS. However, both methods require multiple DBMS instances to test. In contrast, GraphGenie detects logic bugs in a single DBMS instance and has a wider application scenario.

8 Conclusion

We have presented an effective approach, called injective and surjective Graph Query Transformation (GQT), for detecting logic bugs in GDBMSs, which we have implemented as a practical tool called GraphGenie. Our approach leverages graph properties to generate follow-up queries by mutating graph query patterns, resulting in the detection of 25 bugs in mature GDBMSs with positive feedback from the developers. Compared to existing works, GraphGenie is free of false alarms and effectively detects previously unknown logic bugs. Our experimental results also demonstrate that GQT can help find more bug-inducing test cases by providing insights into mutating graph query patterns. Additionally, GraphGenie is scalable and can detect performance issues in GDBMSs. Overall, we believe that GraphGenie is a practical testing tool that can improve the reliability and robustness of GDBMSs.

Acknowledgements

We thank the anonymous reviewers for their suggestions to improve the paper. This research/project is supported by the National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing <NRF-NCR25-Fuzz-0001>). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] Sjoerd Mullender Andreas Seltenreich, Bo Tang. [n. d.]. https://github.com/anse1/sqlsmith.
[2] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Computing Surveys (CSUR)* 40, 1 (2008), 1–39.
[3] Asim Ansari, Skander Essegaier, and Rajeev Kohli. 2000. Internet recommendation systems.
[4] Apache. [n. d.]. https://tinkerpop.apache.org/.
[5] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *The 45th International Conference on Software Engineering (ICSE'23)*.
[6] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.
[7] Bitnine. [n. d.]. http://www.agensgraph.org/.
[8] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. 2007. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation.. In *USENIX Security Symposium*, Vol. 15.
[9] Wing Kwong Chan, Shing Chi Cheung, and Karl RPH Leung. 2007. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research (IJWSR)* 4, 2 (2007), 61–81.
[10] Peter Chapman and David Evans. 2011. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proceedings of the 18th ACM conference on Computer and communications security*. 263–274.
[11] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2002. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2002).
[12] Tsong Yueh Chen, Joshua WK Ho, Huai Liu, and Xiaoyuan Xie. 2009. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC bioinformatics* 10, 1 (2009), 1–12.
[13] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, TH Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and

opportunities. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–27.
[14] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99.
[15] DB-Engines. [n. d.]. https://db-engines.com/en/ranking/graph+dbms.
[16] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph neural networks for social recommendation. In *The world wide web conference*. 417–426.
[17] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-Free DBMS Fuzzing. In *Conference on Automated Software Engineering (ASE'22)*.
[18] HugeGraph. [n. d.]. https://hugegraph.apache.org/.
[19] JanusGraph. [n. d.]. https://janusgraph.org/.
[20] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In *32st USENIX Security Symposium (USENIX Security 23)*.
[21] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2020. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB 2020)*. Tokyo, Japan.
[22] Matteo Kamm. 2022. *Testing Graph Databases using Predicate Partitioning*. Master's thesis. ETH Zurich.
[23] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.
[24] Fei-Ching Kuo, Tsong Yueh Chen, and Wing K Tam. 2011. Testing embedded software by metamorphic testing: A wireless metering system case study. In *2011 IEEE 36th Conference on Local Computer Networks*. IEEE, 291–294.
[25] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. (2014), 216–226. https://doi.org/10.1145/2594291.2594334
[26] Yu Liang, Song Liu, and Hong Hu. 2022. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4309–4326.
[27] Wei Lin, Ziyue Hua, Luyao Ren, Zongyang Li, Lu Zhang, and Tao Xie. 2022. GDsmith: Detecting Bugs in Graph Database Engines. *arXiv preprint arXiv:2206.08530* (2022).
[28] Jiahao Liu, Jun Zeng, Xiang Wang, Kaihang Ji, and Zhenkai Liang. 2022. Tell: log level suggestions via modeling multi-level code block information. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 27–38.
[29] Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. 2023. Learning Graph-based Code Representations for Source-level Functional Similarity Detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 345–357.
[30] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering*. 225–236.
[31] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic testing of Datalog engines. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 639–650.
[32] MarketsandMarkets. [n. d.]. Graph Database Market. https://www.marketsandmarkets.com/Market-Reports/graph-database-market-126230231.html.
[33] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.
[34] MemGraph. [n. d.]. https://memgraph.com/.
[35] Neo4j. [n. d.]. https://neo4j.com/developer/cypher/.
[36] OpenCypher. 2019. Cypher Technology Compatibility Kit Report of Overview of Language Coverage in Cypher-for-Gremlin. https://opencypher.github.io/cypher-for-gremlin/test-reports/1.0.4/cucumber-html-reports/overview-features.html.
[37] Neo4j Graph Platform. [n. d.]. https://neo4j.com/.
[38] Arvind Ramanathan, Chad A Steed, and Laura L Pullum. 2012. Verification of compartmental epidemiological models using metamorphic testing, model checking and visual analytics. In *2012 ASE/IEEE International Conference on BioMedical Computing (BioMedCom)*. IEEE, 68–73.
[39] Redis. [n. d.]. https://github.com/RedisGraph/RedisGraph.
[40] Paul Resnick and Hal R Varian. 1997. Recommender systems. *Commun. ACM* 40, 3 (1997), 56–58.
[41] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Sacramento, California, United States) *(ESEC/FSE 2020)*. https://doi.org/10.1145/3368089.3409710
[42] 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.
[43] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (2020). https://doi.org/10.1145/3428279
[44] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Banff, Alberta.
[45] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. Citeseer, 618–622.
[46] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing database systems via differential query execution. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*.
[47] TinkerPop. [n. d.]. https://tinkerpop.apache.org/gremlin.html.
[48] Website. 1988. TPC-H Benchmark. https://www.tpc.org/tpch/. Accessed: 2023-03-15.
[49] Website. 2013. American Fuzzy Lop (AFL) Fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2023-03-15.
[50] Wikipedia. [n. d.]. Bijection, injection and surjection. https://en.wikipedia.org/wiki/Bijection,_injection_and_surjection.
[51] Rongjing Xiang, Jennifer Neville, and Monica Rogati. 2010. Modeling relationship strength in online social networks. In *Proceedings of the 19th international conference on World wide web*. 981–990.
[52] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.
[53] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.
[54] Jun Zengy, Xiang Wang, Jiahao Liu, Yinfang Chen, Zhenkai Liang, Tat-Seng Chua, and Zheng Leong Chua. 2022. Shadewatcher: Recommendation-guided cyber threat analysis using system audit records. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 489–506.
[55] 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*. 302–313.
[56] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970.