Testing Graph Database Systems via Graph-Aware Metamorphic Relations

Zeyang Zhuang
The Chinese University of Hong Kong
Hong Kong SAR, China
zyzhuang22@cse.cuhk.edu.hk

Penghui Li
The Chinese University of Hong Kong
Hong Kong SAR, China
phli@cse.cuhk.edu.hk

Pingchuan Ma
Hong Kong University of Science and Technology
Hong Kong SAR, China
pmaab@cse.ust.hk

Wei Meng
The Chinese University of Hong Kong
Hong Kong SAR, China
wei@cse.cuhk.edu.hk

Shuai Wang
Hong Kong University of Science and Technology
Hong Kong SAR, China
shuaiw@cse.ust.hk

ABSTRACT

Graph database systems (GDBs) have supported many important real-world applications such as social networks, logistics, and path planning. Meanwhile, logic bugs are also prevalent in GDBs, leading to incorrect results and severe consequences. However, the logic bugs largely cannot be revealed by prior solutions which are unaware of the graph native structures of the graph data. In this paper, we propose Gamera (Graph-aware metamorphic relations), a novel metamorphic testing approach to uncover unknown logic bugs in GDBs. We design three classes of novel graph-aware Metamorphic Relations (MRs) based on the graph native structures. Gamera would generate a set of queries according to the graph-aware MRs to test diverse and complex GDB operations, and check whether the GDB query results conform to the chosen MRs.

We thoroughly evaluated the effectiveness of Gamera on seven widely-used GDBs such as Neo4j and OrientDB. Gamera was highly effective in detecting logic bugs in GDBs. In total, it detected 39 logic bugs, of which 15 bugs have been confirmed, and three bugs have been fixed. Our experiments also demonstrated that Gamera significantly outperformed prior solutions including Grand, GDsmith and GDBMeter. Gamera has been well-recognized by GDB developers and we open-source our prototype implementation to contribute to the community.

PVLDB Reference Format:
Zeyang Zhuang, Penghui Li, Pingchuan Ma, Wei Meng, and Shuai Wang. Testing Graph Database Systems via Graph-Aware Metamorphic Relations. PVLDB, 17(4): 836 - 848, 2023.
doi:10.14778/3636218.3636236

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/cuhk-seclab/Gamera.

Proceedings of the VLDB Endowment, Vol. 17, No. 4 ISSN 2150-8097.
doi:10.14778/3636218.3636236

1 INTRODUCTION

Graph database systems are a typical type of database systems for processing graph data containing nodes and edges. They have powered many applications such as social networks [29], knowledge graphs [23, 40], and fraud detection [36]. GDBs are widely used in practice to enable efficient storage and queries of graph data. For instance, the leading provider of graph technology and the most popular GDB, Neo4j [5, 13], has been used by more than 950 enterprise customers, and supported more than 75% of connected data applications of the Fortune 100 [27].

Similar to other software systems [41, 42], GDBs contain logic bugs, which could cause unexpected behaviors and lead to severe consequences. For example, a logic bug in a social network GDB might cause incorrect friend recommendations, leading to user dissatisfaction. Logic bugs in GDBs refer to the errors or flaws in the way graph data is represented, stored, or queried that lead to erroneous results, *e.g.*, omitting an anticipated node in the query result. These bugs mainly stem from incorrect implementation or optimization [39] of the GDB engines. Since logic bugs usually do not crash the systems, automatically detecting them is inherently challenging. We need testing oracles to verify whether the expected behaviour is observed in each test.

```
1 MATCH path=(a)-[*]-(b) WHERE ID(a)=0 AND ID(b)=1
      RETURN COUNT(path)
2 // -> 0
3 MATCH path=(a)-[*]-(b) WHERE ID(a)=1 AND ID(b)=2
      RETURN COUNT(path)
4 // -> 4
5 MATCH path=(a)-[*]-(b)-[*]-(c) WHERE ID(a)=0 AND ID(b)
      =1 AND ID(c)=2 RETURN COUNT(path)
6 // -> Error 4 != 0*4
```

Listing 1: A logic bug detected in RedisGraph. The number of paths in the third query result is incorrect.

Listing 1 shows a real-world logic bug we detected in RedisGraph [1]. The query in line 1 counts the number of paths between the first node with id 0 and the second node with id 1. Line 3 calculates the number of paths between the second node and the third

[1]The issue can be found at https://github.com/RedisGraph/RedisGraph/issues/2929.

node with id 2. The first two queries get results of zero and four, respectively. It is evident that the number of paths that start from the first node, pass the second node, and finally reach the third node should be identical to the multiplication of these two queries' results, *i.e.*, zero. However, the query in line 5 returns four incorrectly, which indicates a logic bug. The RedisGraph developers have confirmed this bug and are working on fixes at the time of writing.

To detect logic bugs in GDBs, multiple approaches have been proposed. GDsmith [32] and Grand [47] leverage differential testing [38]. They generate syntax-correct queries and feed the same queries to multiple GDBs for checking inconsistent results among the GDBs. The limitations of such differential testing approaches are that all GDBs yielding the same result does not necessarily indicate correctness because the tested GDBs could all suffer from the same bugs, and they can only test the functionalities overlapped across the tested GDBs. Metamorphic testing approaches [26] do not have such a limitation. Metamorphic testing uses a set of carefully-designed Metamorphic Relations (MRs) to mutate test inputs and expose bugs by checking if the outputs violate certain (invariant) properties specified by the MRs. In relational database systems, SQLancer [18] leverages ternary logic partitioning (TLP) [42] and non-optimizing reference engine construction (NoREC) [41] to form MRs and detect logic bugs. To our best knowledge, GDBMeter [34] is the only work adopting metamorphic testing to test GDBs. It ports the idea of TLP to split a given query into three sub-queries, and the MRs it employs specify that the union of the three sub-query result sets should be equal to the original result set.

However, prior metamorphic testing approaches cannot effectively detect many logic bugs (*e.g.*, the one in Listing 1) because they are unaware of the graph native structures. Specifically, the graph data manipulated by GDBs contains complex semantics and relationships among the graph native structures—*i.e.*, nodes, edges and paths—such as various types of directed edges, cycles, *etc.* GDBs thus have specific code logic for handling the graph native structures. However, prior solutions do not analyze the graph native structures and cannot effectively reveal the related logic bugs. They further have two fundamental problems. First, they have *limited support for graph query syntaxes* and are unable to comprehensively exercise the GDBs' operations on the graph native structures. For instance, they even do not support the commonly-used path traversal syntax, *union* clauses, *etc.* They thus cannot thoroughly test the related important functionalities of GDBs. Second, they *lack necessary testing oracles* to effectively identify bugs related to graph native structures. They directly reuse oracles (*e.g.*, TLP) from relational database system benchmarking, which cannot capture the graph native structures in graph data. In order to comprehensively test GDBs, new MRs derived from the graph native structures need to be proposed.

In this paper, we propose Gamera (Graph-aware metamorphic relations), a novel metamorphic testing solution to uncover logic bugs in GDBs. Based on the features of graph native structures, we design a set of new graph-aware MRs and oracles, and support the related query language syntaxes to address the above-mentioned issues. In particular, Gamera employs three classes of novel graph-aware MRs. We first design several *elementary MRs* to test and cover the fundamental graph data operations over the primitive-type graph data. We then construct the *compound MRs* from the elementary MRs for testing more complex GDB functionalities. We design two graph query pattern manipulation techniques to transform the existing basic (or complex) query patterns to equivalent but more complex (or simpler) ones for developing the compound MRs. Finally, we consider the dynamic update of the graph data in the GDBs and propose several *dynamic MRs* that are derived from our novel graph mutation techniques. Based on the three classes of graph-aware MRs, Gamera generates a set of queries and uses oracles to check if the results conform to the chosen MR. In addition, Gamera resolves the syntax support problem by handling a more comprehensive set of graph query syntaxes in graph query languages, such as path traversals, *union* clauses, *etc.*

We implemented a prototype of Gamera with 14,000 lines of code for testing Cypher and Gremlin based GDBs. We thoroughly evaluated the effectiveness of Gamera on seven widely-used GDBs such as Neo4j [13] and OrientDB [15]. At the time of writing, Gamera has detected in total 39 bugs, of which 15 bugs have been confirmed, and three bugs have been timely fixed. The good performance of Gamera resulted from all the three classes of MRs. We found that our compound MRs were the most effective by contributing to the detection of 30 bugs. The elementary and dynamic MRs helped detect five and four bugs, respectively. We also compared Gamera with three state-of-the-art GDB benchmarking tools (GDsmith [32], Grand [47] and GDBMeter [34]) on the same set of GDBs. Gamera was able to detect all the eight and five bugs that Grand and GDBMeter could detect, respectively, and Gamera outperforms GDsmith in detecting logic bugs. This further demonstrates the advantage of our techniques over the prior ones and the importance of employing graph-aware MRs for detecting logic bugs in GDBs. Our automated benchmarking approach received positive feedback from the GDB community and we have publicly released our research artifact of Gamera to further contribute to it.

In summary, our paper makes the following contributions:

- We propose a set of novel graph-aware MRs to facilitate logic bug detection in GDBs.
- We design and open-source a new system, Gamera, for benchmarking Cypher-based and Gremlin-based GDBs.
- With Gamera, we have detected 39 bugs, among which 15 bugs have been confirmed by the developers.

2 BACKGROUND AND MOTIVATION

2.1 Labeled Property Graph in GDBs

Labeled property graph model [44] is commonly used by graph database systems (GDBs) such as Neo4j, RedisGraph, OrientDB, and JanusGraph to represent their graph structures. It contains a set of nodes and a set of edges associated with those nodes. Each node or edge has an attached label to separate it into a specific group. A set of key-value pair attributes is used to describe the properties and provide additional metadata of the nodes or edges.

Figure 1 shows an example of a labeled property graph, which consists of two nodes and one edge. Specifically, a node (v:1) with label *person* has properties *name* and *age*, while a node (v2) with label *software* has properties *name* and *lang*. One directed edge (e:1) labeled by *created* has the *since* property, which indicates when a person created the software. In this paper, we define the nodes, edges, and paths of labeled property graphs in GDBs as *graph native*

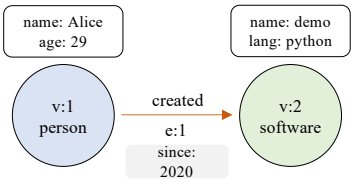


Figure 1: A labeled property graph. A unique identifier is assigned for each node and edge, *e.g.*, v:1 represents a node with an id 1.

structures. To guarantee the fundamental functionalities, GDBs need to naturally handle such graph native structures in graph data.

2.2 Graph Query Languages

Unlike relational database systems which support the standardized Structured Query Language (SQL), different GDBs support distinct graph query languages. GDBs utilize these graph query languages to create, access, and modify graph data. For example, Neo4j [13] develops Cypher [30]; TinkerPop [21] develops Gremlin [43]; TigerGraph [19] develops GSQL [28]; and ArangoDB [3] develops AQL [2], *etc.* Specially, most of the GDBs use Cypher and Gremlin for graph data management [5]. Therefore, in this paper we mainly focus on Cypher-based and Gremlin-based GDBs. We introduce the main syntaxes of these two popular languages as follows.

Cypher. Cypher is a declarative and SQL-like query language that allows for matching nodes and edges in labeled property graphs. Cypher uses the ASCII-art syntax. For instance, it represents nodes using round brackets and edges using arrows. Here we give an example of Cypher query: `MATCH (p:person)-[:created]->(:software ) WHERE p.age > 25 RETURN p.name}`. It returns the names of persons who are over the age of 25 and have created the software. In this query, *MATCH* is an operator usually used to describe the subgraph for finding nodes, edges, or paths; *WHERE* is used to add additional constraints to the subgraphs and filter out unwanted ones; and *RETURN* specifies how the results should be outputted. Cypher provides various syntaxes and constraints [4] for querying graph data, including specifying patterns, filtering patterns, indexing, aggregation, *etc.*

Gremlin. Gremlin is a functional language whereby traversal operators are chained together to form path-like expressions. A Gremlin query consists of a sequence of Gremlin traversal primitives [10], which ultimately calculate the final outputs. We provide an example of Gremlin query here: `g.V().where(values('name').is(eq('Alice') )).outE('created').inV().hasLabel('software').values('lang')`. This query first gets all nodes using *g.V()*. It then filters the nodes whose name equals to Alice using *where()* and *eq()*. The software created by Alice can be accessed by traversing the edges using *outE('created').inV()*. Finally, it returns the language of the software by *hasLabel()* and *values()*. In this example, a nested query exists in the *where()* API, in which *is()* is used to assess whether a property value matches the predicate. The traversal style of Gremlin is noticeable in this case since we retrieve the final results by calling a sequence of functional APIs.

Table 1: Comparison of existing approaches.

	Type	Graph Query Lang.	Lang. Syntaxes	Graph Native Structure	Graph-aware MRs
GDsmith [32]	DT	Cypher	◐	○	○
Grand [47]	DT	Gremlin	◐	○	○
SQLancer [18]	MT	○	○	○	○
TLP [42]	MT	○	○	○	○
GDBMeter [34]	MT	Cypher & Gremlin	◐	○	○
Gamera	MT	Cypher & Gremlin	●	●	●

○/◐/● denote a tool is incapable/partially capable/fully capable.

2.3 Existing Approaches and Limitations

We summarize existing GDB benchmarking solutions in Table 1. Some prior approaches such as GDsmith [32] and Grand [47] employ differential testing (DT). They feed the same queries into multiple GDBs, aiming to find inconsistent results among them. However, all GDBs returning the same result does not necessarily imply correctness. Using inconsistency as the oracle in DT inherently brings two issues: 1) it cannot detect the bugs that exist in all the tested GDBs as they would always return the same incorrect results [47]; and 2) it can only test the common functionalities overlapped across the tested GDBs but not the unique functionalities of a specific GDB.

Metamorphic testing (MT) [26, 37] alleviates the (differential) testing oracle problem with Metamorphic Relations (MRs). To test a program, MT mutates program inputs and detects bugs by checking if the outputs violate certain (invariant) properties specified by the MRs. As an example, to test the implementation of $\sin(x)$, MT asserts whether the MR $\sin(x) = \sin(\pi - x)$ always holds when arbitrarily mutating x. A bug in $\sin(x)$ is detected when input x and its variant $\pi - x$ produce inconsistent outputs. MT thus does not require the existence of multiple similar program implementations (*e.g.*, multiple database systems in GDB testing). To date, MT methods have found hundreds of bugs in database systems, demonstrating its effectiveness [18, 41, 42]. However, most prior database MT works focus on relational database systems [18, 41, 42]. Due to the huge differences between relational and graph database systems (*e.g.*, query languages, structures of stored data, *etc.*), most of them cannot be directly applied to the domain of GDBs. To the best of our knowledge, GDBMeter [34] is the only MT tool for GDBs. It ports the idea of ternary query partitioning [42] used in relational databases to GDBs. It splits a given query into three derived subqueries, in which the predicates are evaluated to TRUE, FALSE, and IS NULL, respectively. It then validates the consistency between the union of the derived result sets and the original one.

GDBs are designed and expected to properly handle the graph native structures for manipulating the graph data. However, existing MT solutions are unaware of graph native structures in GDBs; this hinders their effectiveness. The reasons are two-fold. First, graph query languages of GDBs contain a comprehensive set of language syntaxes related to the graph native structures, whereas prior MT approaches only *cover limited syntaxes*. For instance, the commonly-used path traversal syntax and *union* clauses are not supported by prior works. As a result, they cannot well test all the fundamental operations in GDBs and reveal related logic bugs (more details are in §6.3).

Second, prior MT approaches *lack necessary testing oracles* to effectively detect bugs related to graph native structures. Given

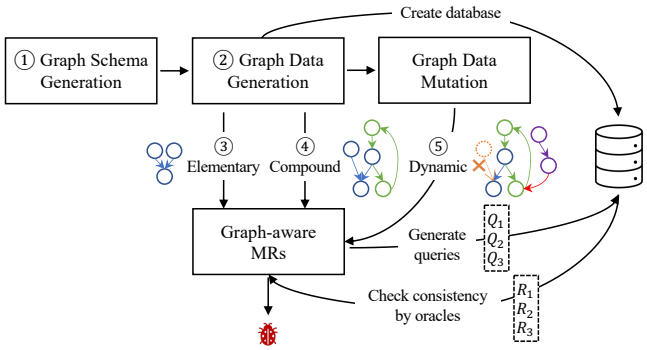


Figure 2: The workflow of GAMERA.

the unique features of graph data in GDBs, existing testing oracles cannot sufficiently identify related bugs like the one in Listing 1. For instance, GDBMeter directly reuses the existing ternary query partitioning MR from relational database systems but does not analyze the unique graph native structures. Therefore, it cannot thoroughly test GDBs. Without testing oracles associated with the graph native structures, lots of severe logic bugs would not be identified. Defining oracles, however, is a non-trivial and well-known research challenge in MT.

3 OVERVIEW

In this paper, we aim to advance the benchmarking technique of graph database systems and address the limitations of the prior works. We propose GAMERA, an effective and automated metamorphic testing solution to detect logic bugs in graph database systems. We tackle the first limitation by supporting more graph query syntaxes (*e.g.*, path traversals, *union* clauses, *etc.*) for the two mainstream graph query languages—Cypher and Gremlin. Besides, we resolve the second limitation by proposing a set of novel graph-aware MRs in GAMERA. The new MRs perceive the graph native structures and features to define effective testing oracles. These MRs are categorized into three classes, which could comprehensively test the various functionalities of GDBs. The comprehensive language syntax supports and the new graph-aware MRs enable GAMERA to expose new classes of logic bugs related to graph native structures, which cannot be found by the prior solutions.

Figure 2 shows the workflow of GAMERA. In ①, GAMERA generates the graph schema to define the labels of nodes and edges, as well as their property names and corresponding types. In ②, it randomly creates graph data based on the schema. The graph data is inserted by a sequence of random graph queries (*e.g.*, create and update queries) to create the graph database. According to the generated graph data, GAMERA refers to our novel graph-aware elementary MRs (③), compound MRs (④), and dynamic MRs (⑤) for testing GDBs. In each test, it generates a set of queries (*e.g.*, Q_1, Q_2, Q_3) based on the graph-aware MRs. Finally, GAMERA uses the corresponding oracles to label and report bugs. The oracle checks if the result sets R_1, R_2, R_3 produced by these queries conform to the selected MR. A consistency violation of such an MR indicates a logic bug in the GDB.

We further use the example in Listing 1 to demonstrate the workflow of GAMERA for detecting logic bugs in GDBs. First, GAMERA randomly generates a graph schema and graph data with 100 nodes and 200 edges. The number of nodes and edges can be customized. It then generates three testing queries according to the MR shown in Listing 1. Specifically, Q_1 (line 1), Q_2 (line 3), and Q_3 (line 5) are generated to count the number of paths between nodes. Q_3 is the corresponding oracle statement, which checks whether R_3 (line 6) is consistent with the result computed from R_1 (line 2) and R_2 (line 4). As there is an MR violation, GAMERA reports it as a bug.

In the following content, we describe how GAMERA generates the graph schema (①) and the initial graph data (②). In §4, we elaborate on our core design of different graph-aware MRs, and introduce each query generation based on the MRs and the corresponding oracle in detail.

Graph Schema Generation. A graph schema defines the types of nodes and edges as well as their properties in a labeled property graph. By enforcing graph schema, GDBs ensure that only valid queries can be executed on the data. The graph schema guarantees that the queried nodes, edges, and properties actually exist and are correctly referenced. It prevents executing queries that reference nonexistent nodes, edges, or properties, thus avoiding meaningless queries. For example, for the graph data shown in Figure 1, the query statement querying the nodes with *system* label would be blocked from execution by GDBs; GDBs will warn the user that the query structure is incorrect as it violates the graph schema.

In the labeled property graph model, a node contains a label and a set of properties. We represent a node V as $<L_V, \mathcal{P}_V>$, where L_V describes the label for the node and $\mathcal{P}_V$ describes its corresponding properties. A directed edge contains a label, a set of properties, a starting node, and an ending node. Similarly we represent a directed edge E as $<L_E, \mathcal{P}_E, startV, endV>$, where L_E describes the label for the edge and $\mathcal{P}_E$ describes its properties, and $startV, endV$ belongs to the node set V. Concretely, L_V, L_E are randomly generated strings; and $\mathcal{P}_V, \mathcal{P}_E$ are sets composed of elements $<propertyName, T>$, where $propertyName$ and T represent the property name and data type (*e.g.*, integer, float, strings, booleans, and points), respectively. For instance, the node v:1 in Figure 1 can be represented as $<person, «name, String>, <age, Integer»>$. Following the above definitions, GAMERA would randomly generate a set of graph schemas in the GDBs.

Initial Graph Data Generation. Given a graph schema, GAMERA further generates the corresponding graph data that includes n nodes and m edges, where n and m are configurable parameters. Specifically, we first create n nodes. Then, among all the potential directed edges between any two nodes, we randomly create m edges from $|n \times n|$ possible edges. To create a node or an edge, we assign a label as well as the properties selected from the available properties set in the graph schema to them. The properties are name-value pairs in a set. The property value is randomly generated based on a property's type. For each edge, we also specify its starting and ending nodes. We implement the graph data generation process by automatically submitting a series of graph queries (*e.g.*, creation and updating queries) to the GDB with the corresponding graph schema. The graph data generation procedure is randomized and would generate a variety of diverse complex graphs (*e.g.*, graphs with different average degrees).

Table 2: Overview of graph-aware MRs.

Class		MRs
Elementary	Node & Edge Levels	Spouse / Ancestor / Descendant Nodes K-hop Neighbours Edge Properties
	Path Level	Connectivity Path Number Calculation Shortest Path
Compound		Pattern Fusion Pattern Partitioning
Dynamic		Data Addition / Deletion / Update Irrelevant Data Manipulation

4 GRAPH-AWARE MRS

We aim to test the most fundamental and commonly-used operations in GDBs, and reveal logic bugs in them. Since GDBs store and manage graph data in its native structures, the native structures of graph data allow us to derive a set of new graph-aware MRs (shown in Table 2).

We first propose the *elementary graph-aware MRs* to test the fundamental functionalities of GDBs. The graph native structures consist of several graph primitive types such as nodes, edges, and paths. Support for manipulating these graph primitive types is the most fundamental and important operation of GDBs. The elementary MRs can assess the correctness of these functionalities. Next we propose the *compound graph-aware MRs* to test and stress more complex functionalities in GDBs. The interactions between the graph native structures are diverse and complex. For example, the varied distribution of nodes and edges reveals different patterns of behaviors in the graph—that is, graphs contain diverse clusters or communities. Nodes can involve multiple types of directed edges and also include cycles. Besides, graphs have various types of path connectivity patterns such as dense or sparse graphs, and regular or irregular graphs. Checking the connectivity (*i.e.*, confirming whether a path exists) involves many path traversal algorithms [16]. Based on the complex intrinsic interactions of the graph native structures, the compound MRs are proposed correspondingly to ascertain their correctness. Moreover, we propose the *dynamic graph-aware MRs* to test the GDB's functionalities to support dynamic data updates. The graph data in GDBs can be dynamic, meaning that they can change over time because of graph queries. This can lead to complex and unpredictable behaviors, as nodes can form new edges and existing edges can break down. The dynamic MRs can verify whether these work perfectly.

In the rest of this section, we first present the elementary MRs testing fundamental graph data operations in §4.1, then compound MRs that exercise complex GDB operations in §4.2, and finally the dynamic MRs that are derived from our novel graph mutation techniques in §4.3.

4.1 Elementary MRs

We first define two types of elementary graph-aware MRs to detect logic bugs in the fundamental GDB operations for manipulating graph data. We design comprehensive query patterns to cover most fundamental GDB operations. Based on the graph data type in the queries, the elementary MRs are categorized into node and edge level MRs, and path level MRs. In the rest of this section, we elaborate on each type of the elementary MRs.

We use the following definition throughout this section. Let $G = (V, E)$ denote a directed graph G, where V is the vertex set and E is the edge set. Let a vertex $v_i \in V$, and an edge $e = (v_1, v_2) \in E$, indicating that there is a directed connection from v_1 to v_2.

4.1.1 Node & Edge Level MRs We first develop MRs to test GDB functionalities related to node and edge operations. As nodes and edges are the two fundamental and related types of data in graphs, they are commonly manipulated together and the operations on nodes and edges are essential to all other functionalities of GDBs. Therefore, we discuss node and edge level MRs together.

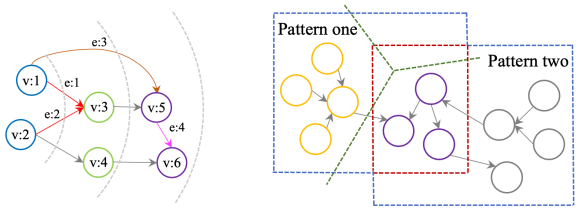
Node Level MR. We first propose several MRs that test the relationship between nodes. As shown in Table 2, we cover common node relationships, including spouse, ancestor, and descendant relationships.

Spouses are nodes that have the same child node(s), *i.e.*, they have outgoing edges connecting to the same child node(s). In Figure 3(a), we can see that v:1 and v:2 are spouses. We then can develop the spouse MR with the following queries. First, we obtain the set $spouses(v_A)$ of all spouse nodes of node A; for any node B in $spouses(v_A)$, we obtain its spouses as in the set $spouses(v_B)$. Then node A must be a member of the set $spouses(v_B)$, which can be validated by the oracle for reporting bugs. A violation of the inclusion relation between the node and the spouse set implies a logic bug.

Ancestors and descendants of a node would also be commonly queried. If one can follow a sequence of directed edges from a node A to reach a node B, then node B is a descendant of node A, which is an ancestor of node B. The ancestral and descendant relationships also imply that the ancestor and descendant are connected. In Figure 3(a), we can see that v:1 is the ancestor of v:3, while v:5 is the descendant of v:2. Similarly, we develop the ancestor/descendant MR as follows. The oracle can check that if node A is inside the set $ancestors(v_B)$, then node B must be in the node set $descendants(v_A)$, and vice versa.

K-hop Neighbour MR. K-hop neighbourship is another classic relationship among nodes and edges. The k-hop neighbour nodes of a node A are the set of nodes that can be reached from node A by going through k directed edges in either incoming or outgoing direction. For example, in Figure 3(a), node v:5 is an outgoing 2-hop neighbour node of node v:2. The k-hop neighbours of node A can be formally defined as: $neighbours(v_A, k) = \{v \in V \mid distance(v_A, v) = k\}$. Similarly, we can develop an MR with the following queries. If node B is inside the outgoing k-hop neighbours $neighbours(v_A, k)$ of node A, then the oracle can check that node A must also be inside the incoming k-hop neighbours $neighbours(v_B, k)$ of B. If node A is omitted from this set, then a logic bug is found.

Edge Property MR. We next design MRs to test the relationships between edges. In particular, we design edge property MR on top of node level MR and k-hop neighbour MR to increase the complexity of the queries. We consider two types of edge relationships. We define edges with the same property name as *homogeneous edges*, and edges with different property names as *heterogeneous edges*. In Figure 3(a), we can see that edges e:1, and e:2 in red color represent



(a) Elementary MRs (b) Compound MRs

Figure 3: The left and the right are the sample graphs related to elementary MRs and compound MRs, respectively. Specifically, in Figure 3(b), the area circled by the blue lines represents the union of the two patterns. The red rectangle represents the intersection of the two patterns. The green lines partition pattern one into three sub-patterns.

edges with the same property, and edges (*e.g.*, e:1, e:3, and e:4) in different colors represent edges with diverse properties.

When traversing a graph to find the nodes' spouses, ancestors or descendants, and k-hop neighbours in our design, the above node level MR and k-hop neighbour MR do not specify the properties of the edges. By considering the edge properties, we can further select homogeneous edges of the same property or heterogeneous edges of a selected set of properties during the graph traversal to increase the complexity of node level MR and k-hop neighbour MR. Therefore, these edge property MRs would be able to cover more code related to edge property processing in the GDBs. We use the following queries as an example. First, we obtain the set $spouses'(v_A)$ of all spouse nodes connected to node A by the edges of the selected properties; for any node B in $spouses'(v_A)$, we obtain its spouses with the same selected properties of the edges as in the set $spouses'(v_B)$. Then node A is expected to be within the set $spouses'(v_B)$ and the oracle can check that.

4.1.2 Path Level MRs The GDBs implement diverse path finding algorithms [16, 22] for retrieving paths in the graphs. The correctness of such operations is important to a lot of computation tasks such as logistics and navigation. We design several path level MRs to test the correctness of those relevant implementations in the GDBs.

We first test the ***connectivity*** in the GDBs. The connectivity between two nodes indicates whether there exists at least one path between the two nodes in the graph. Our intuition is that two nodes would be connected if they can be connected via another node. Therefore, we develop a set of connectivity queries according to this connectivity MR. Specifically, if a node A is connected to node B, and node B is connected to node C, then the oracle can check that there exists at least one path from node A to node C, regardless of whether there is a directed edge from A to C or not. For instance, as shown in Figure 3(a), v:2 is connected to v:4 and v:4 is connected to v:6, therefore v:2 should also be connected to v:6. Note that since the edges (hence paths) are directed, the connectivity is also directional—A being connected to B does not necessarily indicate that B is connected to A.

We next test the ***path number calculation*** in the GDBs. Computing the number of paths between any two nodes is also a frequently used feature in GDBs. To test such a functionality, our core idea is to develop queries that directly and indirectly calculate the number of paths between two nodes and then ask the oracle for consistency check. We consider three randomly chosen nodes—A, B and C; we calculate the number of paths from A to B as x, the number of paths from B to C that do not include A as y, and the number of paths from A to C via B as z. Specifically, the paths are all simple paths. Then the oracle would be able to tell whether z is the product between x and y and help report inconsistencies as bugs. It is important to note that we do not query *all* the paths from A to C as they may or may not contain the node B, which makes it difficult to establish a deterministic metamorphic relation. As we already discussed an example in Listing 1, the number of paths that start from v:0, pass v:1, and end at v:2 should be zero, which equals the first result (zero) multiplied by the second result (four). The transit node v:1 is necessary for establishing the relationship. Our evaluation results show that such an MR facilitates uncovering several errors in complex and large-scale graphs like RedisGraph [17] and OrientDB [15].

We finally propose an MR to test the ***shortest path*** implementation in the GDBs. Searching the shortest path between two nodes is also a critical operation that is widely used in many application scenarios such as route planning, logistics, navigation, *etc.* Similarly, we consider the shortest paths related to three nodes in the graph. Our intuition is that the shortest path between two nodes would not become shorter if a critical node on the original shortest path is removed. Specifically, we first query the shortest path p from node A to node B and pick an intermediate critical node C on the path p. Next we exclude node C from the graph and query the new shortest path p' from A to B. Then the oracle would check if p is not longer than p' by evaluating $length(p) \leq length(p')$. In Figure 3(a), we can see that the length of the shortest path from v:2 to v:6 is 2. After excluding v:4, the shortest path's length becomes 3, which is greater than or equal to 2.

4.2 Compound MRs

We have covered the elementary graph-aware MRs, which we propose to test the fundamental GDB operations over graph data. However, the elementary MRs can test only the individual basic graph manipulation functionalities, *e.g.*, querying for a small number of certain nodes, edges, or paths. Many logic bugs would be manifested only when the GDBs are stressed with more complex operations. Therefore, we further propose several compound MRs for testing more complex GDB functionalities.

We construct the compound MRs from the elementary MRs by generating more complex graph query patterns from the simpler basic query patterns in the elementary MRs. We propose two graph query pattern manipulation techniques (*i.e.*, MRs) to transform the existing basic (or complex) query patterns to equivalent but more complex (or simpler) ones. These techniques allow us to create new complex MRs from the existing ones and test complex GDB operations.

4.2.1 Query Pattern Generation We first discuss how to generate some basic graph query patterns for later generating more complex patterns that can exercise the complex functionalities of GDBs. The basic patterns would serve as the building blocks for

producing more complex query patterns using our pattern manipulation techniques. We define a graph query pattern as a graph matching pattern for querying primitive graph data such as nodes, edges, or paths. We develop two methods to generate the basic query patterns.

First, we generate basic query patterns directly according to the elementary MRs we earlier defined. In the elementary MRs, the involved query patterns instruct the GDBs to perform the basic graph data operations over the primitive types of graph data. For instance, such basic query patterns can query the neighbour or spouse nodes of a node, or query the (shortest) paths between any two nodes.

Second, we generate basic query patterns that query the GDBs for some random portion of the graph data with randomly constructed matching criteria. For example, we may randomly specify some matching rules on the properties of the nodes or edges, or add constraints (*e.g.*, including or excluding some specific transit nodes) to the paths between some nodes. By combining multiple random matching criteria, the generated query patterns would invoke quite complex operations in the GDBs.

4.2.2 Query Pattern Transformation We further propose two query pattern manipulation techniques to generate complex query patterns for triggering more code in GDBs as shown in Figure 3(b). The query pattern manipulation techniques can transform existing query patterns into different but equivalent ones, thus generating new MRs that can help uncover logic bugs related to the corresponding complex operations.

We first propose the ***pattern fusion*** technique to generate compound MRs. It fuses two or more graph query patterns into a more complex one with logical operations such as *and* and *or*. Therefore, the query result of the fused pattern would be the same as the result computed by applying the same logical relationship to the query results of the original patterns. Specifically, given two query patterns Q_1 and Q_2 and their query results R_1 and R_2, we compute a fused query pattern Q_3 by using logical *and* or *or* operations and obtain its query result R_3. The oracle could check if R_3 is the *intersection* or *union* of R_1 and R_2 and report inconsistencies as bugs. For instance, Q_1 queries node A's spouses as R_1 and Q_2 queries its 3-hop neighbour nodes as R_2; applying the *and* fusion we derive Q_3 that queries nodes that are both the spouses and 3-hop neighbours of node A as R_3. Then it is expected that R_1 intersects R_2 should equal to R_3.

The pattern fusion technique would allow us to generate many complex MRs, which greatly facilitate testing. We currently limit fusion on patterns querying for the same types of graph data as the logical operation should be applied to query results in the same type. It is important to note that the pattern fusion transformation can be applied to more than two query patterns to construct even more complex queries and MRs.

We next propose the ***pattern partitioning*** technique to generate compound MRs by extending the query partitioning technique in the literature [34, 42]. Contrary to the pattern fusion technique that combines simpler patterns to a complex one, the pattern partitioning technique divides one complex pattern into several simpler sub patterns. Similarly, the query result of the original complex pattern would be related to those of the divided simpler sub query patterns. Specifically, given an original complex query pattern Q, we divide it into three sub query patterns—Q_1, Q_2 and Q_3, of which the predicates are evaluated to *TRUE*, *FALSE*, and *IS NULL*, respectively. For the three sub queries, we generate three predicates: p, *NOT* p, and p *IS NULL*. Each predicate is used in a filter clause to partition the original pattern result set. For example in Figure 3(b), p is used to match the nodes in orange color, then *NOT* p is used to match the nodes in non-orange color (*i.e.*, the purple nodes), and p *IS NULL* returns empty. These three return sets form the complete set of pattern one. This partitioning transformation then leads to a new compound MR: Q's query result R would be the union of the query results (*e.g.*, R_1, R_2 and R_3) of the three sub patterns, *i.e.*, $R = R_1 \cup R_2 \cup R_3$. Specifically, our technique extends the previous work [34]. As elaborated earlier, we generate more comprehensive query patterns, for example, we support queries with complex path retrieval and new syntaxes *etc.* Thus this pattern partitioning technique allows us to generate many complex MRs to stress the GDB.

4.3 Dynamic MRs

In this section, we introduce several dynamic MRs by considering the dynamic update of the graph data in the GDBs. The prior elementary and compound MRs mostly query over the static initial data we randomly generated. In other words, the queries would only cover GDB functionalities for searching and retrieving static data. However, real-world GDB workloads would involve frequent data updates, including the addition, deletion, and modification of graph data. Furthermore, GDBs support much more flexible data schemas to allow diverse data update operations, compared to relational databases, which follow certain constraints (*e.g.*, primary key constraint, foreign key constraint, *etc.*) to update data. Therefore, we also design new dynamic MRs for testing the correctness of GDBs that support dynamic data updates.

We design several *graph data mutation* techniques to update the graph data, and accordingly develop new dynamic graph-aware MRs between queries before and after the data mutation. Our main idea is that the data mutation would result in either expected changes or no change in the query results of the same queries. By submitting the queries before and after certain graph data mutation, we would be able to tell if the new query result is correct or not and thus detect logic bugs. Based on how the data can be updated, we specifically propose the following four graph data mutation techniques and the corresponding dynamic MRs. Note that the dynamic MRs do not require changes of the query patterns and we can reuse the query patterns generated in the elementary or compound MRs.

Data Addition. The first technique mutates the graph data by inserting new data (*e.g.*, nodes, edges, or properties) into the graph. The data addition mutation would lead to several MRs where the new query results would be updated accordingly. We illustrate some examples below. First, given a query pattern, we insert data that matches the pattern, so that the added data would be returned as part of the new query result after the data addition. Second, given two existing nodes that are not connected, we insert new edges between them or between other intermediary nodes, so that a path between the two nodes would be found after the data addition

whereas the same query for finding the path would return nothing previously. Third, given two connected nodes A and B and another node C that is isolated from the two, we connect an edge between B and C, so that any path from A to C would then include a path from A to B as a prefix after the data addition.

Data Deletion. The second technique mutates the graph data by deleting existing data (*e.g.*, nodes, edges, or properties) from the graph. Similarly, we could propose several dynamic MRs where the query results would change after the deletion mutation. We also discuss some examples next. First, given a query pattern, we delete existing data that matches the pattern, so that new query result should not include the deleted data after the data deletion. Second, given two connected nodes, we delete all the edges connecting them directly or indirectly, so that no path between the two nodes would be found after the data deletion. Third, given two connected nodes, we delete some of the edges or nodes on the paths between them, so that fewer paths between them would be returned after the deletion mutation.

Data Update. The third technique mutates the graph data by updating existing data (*i.e.*, the node or edge properties) in the graph while considering the immutability of labels for nodes and edges in GDBs. Similarly, we propose several dynamic MRs to update the graph data. After applying these MRs, the query results will change accordingly. First, given a query pattern related to specific properties, we update the properties of some returned results to other irrelevant values, so that the number of returned results decreases after the data update. Second, by updating the properties of other nodes or edges to the values of the properties in the returned results, the number of new query results would increase.

Irrelevant Data Manipulation. In addition to developing MRs where the query results are changed after data mutation, we could also design MRs where the query results remain the same regardless of data updates. To this end, the fourth technique still mutates the graph data by either addition, deletion, or update of data that is irrelevant to the selected query patterns. Since the data update is irrelevant to the query patterns, the query results should not be affected by the update. For example, given a query pattern, we add, delete, or update some redundant and unrelated nodes or edges, and yet these data would not be returned as part of the query results.

5 IMPLEMENTATION

We implemented a prototype of Gamera for two mainstream graph query languages—Cypher and Gremlin. The techniques proposed in this work are generic and can be naturally ported to other languages such as GSQL and AQL with only engineering efforts. For graph schema and graph data generation, we generate a customized number of nodes and edges in each test. We used prior approaches to generate valid Cypher and Gremlin queries. For Cypher language, we followed GDsmith [32] and used its skeleton-based and completion technique to generate queries. For Gremlin language, we used Grand's abstract Gremlin traversal model to construct valid Gremlin queries [47]. We then integrated our MRs in the query generation strategies and significantly advanced these approaches in supporting syntaxes (*e.g.*, path traversal syntax, *union* clauses, *etc.*) related to graph native structures. Based on that, we developed our oracles to check the corresponding results.

We implemented Gamera in about 7,000 lines of code (LoC) for each query language, in total 14,000 LoC. The implementation of graph-aware MRs took about 4,000 LoC for each language. We have made our prototype implementation of Gamera publicly available as a contribution to the GDB community.

Table 3: The basic information of the GDBs evaluated in this work. Ranks refers to the ranking of this GDB in DB-Engines Ranking [5]. LoC refers to the total lines of code calculated by cloc [1]. All numbers are the latest as of March 2023.

GDB	Query Lang.	Version	Rank	Github Stars	LoC
Neo4j	Cypher	5.4.0	1	11.2k	663k
RedisGraph	Cypher	2.10.9	-	1.8k	654k
OrientDB	Gremlin	3.2.14	5	4.6k	582k
JanusGraph	Gremlin	0.6.2	7	4.8k	72k
HugeGraph	Gremlin	0.12.0	22	2.2k	89k
TinkerGraph	Gremlin	3.6.2	26	1.7k	532k
ArcadeDB	Gremlin	22.12.1	28	291	179k

6 EVALUATION

To evaluate the effectiveness of Gamera in finding real-world logic bugs, we apply Gamera on a list of popular GDBs. In this section, we first introduce the evaluation setup in §6.1, then analyze the results of bug detection in §6.2, next compare Gamera with other existing works in §6.3, and finally list several case study in §6.4.

6.1 Evaluation Setup

Testing GDBs. Our implementation of Gamera currently supports benchmarking Cypher and Gremlin based GDBs. We thus include seven GDBs into our evaluation dataset, including two Cypher-based GDBs, *i.e.*, Neo4j [13], RedisGraph [17], and five Gremlin-based GDBs, *i.e.*, OrientDB [15], JanusGraph [12], HugeGraph [11], TinkerGraph [20], and ArcadeDB [14]. As shown in Table 3, these GDBs are popular. According to DB-Engine Ranking of Graph DBMS [5], these GDBs are among the top 30 GDBs. Their repositories have received several hundreds to thousands of stars on GitHub. For instance, Neo4j is the market leader, as it is the most widely deployed graph data platform. RedisGraph is an extension of the well-known NoSQL database Redis. It is designed with in-memory architecture to support graph queries with high performance. Specifically, three GDBs (*i.e.*, JanusGraph, HugeGraph, and TinkerGraph) encapsulate a Gremlin server in their own servers, while the other two OrientDB and ArcadeDB implement their own TinkerPop3 [21] interface. These GDBs are generally complex and have a large codebase ranging from 72k LoC to 654k LoC. They have been well-tested in prior works [32, 34, 47].

Testing Method. Gamera provides customized parameter configurations, where users can set the number of nodes and edges for each graph data generation, as well as the number of testing rounds and query groups generated after a certain MR is selected. Each graph data generation is randomized under the constraints of the graph schema, thus generating a wide variety of complex graphs. Each query group consists of multiple queries designed according to the MR, and each query group matches different subgraphs to

Table 4: Bugs found by Gamera. The numbers of detected, confirmed, and fixed bugs are listed.

GDB	Detected	Confirmed	Fixed
Neo4j	10	1	0
RedisGraph	9	6	2
OrientDB	3	3	0
JanusGraph	5	1	0
HugeGraph	7	2	0
TinkerGraph	4	1	1
ArcadeDB	1	1	0
Total	39	15	3

Table 5: Bug categorization by different classes of graph-aware MRs. Node & Edge, and Path refer to the node & edge level MRs, and path level MRs, respectively. Compound refers to the compound MRs and Dynamic represents the dynamic MRs.

GDB	Node & Edge	Path	Compound	Dynamic
Neo4j	1	1	6	2
RedisGraph	1	1	5	2
OrientDB	0	1	2	0
JanusGraph	0	0	5	0
HugeGraph	0	0	7	0
TinkerGraph	0	0	4	0
ArcadeDB	0	0	1	0
Total	2	3	30	4

enhance the testing variability. In our experiment, we configure Gamera to run 10 rounds under each graph-aware MR of different classes on every benchmarking GDB. In each testing round, we create a graph database with 100 nodes and 200 edges, and 1,000 groups of queries are generated to test the GDB. Since the *data deletion MR* in the dynamic class needs to delete nodes or edges, the size of graph data will be significantly reduced after several rounds of testing, making it easy to return empty results. Therefore, we design 1,000 rounds of testing for *data deletion MR*, and each round of testing will generate a new graph and 10 groups of queries, which mitigates the above problem. Gamera finished all the testing on all GDBs within a total of 48 hours. Gamera records and reports a bug if the result does not conform to the chosen metamorphic relation. For each reported bug, we manually analyze it to confirm whether it is a real logic bug (true positive). Additionally, we filter out duplicate test cases that trigger the same issue.

The choices of using 100 nodes and 200 edges are made from our preliminary quantitative study. Specifically, we first evaluate Gamera with different sizes of the generated graphs. The results show that augmenting the graph size has negligible impact on the bug detection capability; rather, larger size prolongs the execution time for each testing round. Larger size also increases the complexity of bug-triggering proof of concept (PoC) inputs, making it challenging for GDB developers to diagnose and reproduce the bugs. Therefore, we choose the same testing graph size as other research [47] for the experiments.

Testing Environment. We used a machine with a 2.6 GHz 6-Core Intel Core i7 CPU and 32 GB of memory running macOS 13.2.1 for our experimental benchmarking. To run Gamera, we used Java 11 and Apache Maven for project management. All the testing GDBs are installed on the same machine.

6.2 Bug Detection

Our experiments show that logic bugs are widespread in real-world GDBs and Gamera is highly effective in finding them. As shown in Table 4, Gamera has detected a total of 39 bugs on these well-tested GDBs. We responsibly reported all identified bugs to the corresponding developers or vendors. At the time of writing, 15 of the 39 bugs have been confirmed by the developers. Specifically, three of these bugs have been fixed, and the developers promised to patch them in the next versions. The remaining bugs are on the way to be confirmed. Since some logic bugs are complex or related to the internal features of the GDBs, it will take some time for the developers to locate their root causes. We are still proactively communicating with developers to help them reproduce and fix the bugs.

Impacts of Gamera. Gamera has made real-world impacts to the GDB community. For example, the developers of TinkerGraph emphasized the significance of Gamera: *"That's very interesting. We have looked into an automatic and effective GDBs benchmarking technique from time to time. Some folks here will also find it interesting. If it will eventually be made open source, it could have an open-source home here at Apache TinkerPop someday."*. Besides, the developers of HugeGraph acknowledged our findings and actively confirmed that they will patch them in the next release. The developers of OrientDB also marked the bugs and added them to their milestone version. Given the optimistic feedback, we believe that our novel graph-aware MRs and prototype implementation of Gamera would benefit the GDB community in the long term. We thus are enthusiastic about open-sourcing Gamera and contributing to the community to help build correct and robust GDBs.

Efficacy of MRs. *The MRs proposed in this work are essential for revealing these new bugs.* We classify the bugs into categories based on the class of graph-aware MRs Gamera uses to trigger the bugs and show the details in Table 5. Specifically, if the simplified test case that triggers the bug is closely related to a certain class of MRs, we classify the logic bug into this category. Among the seven GDBs, the elementary MRs trigger five bugs, including 3 path retrieval bugs and 2 node and edge matching bugs. We find that some GDBs would calculate the incorrect number of paths when they traverse some complex graphs. The result of relatively few bugs is reasonable because GDBs normally have been well-tested regarding such basic features of the graph structure.

Furthermore, we find that the majority (30 out of 39) of bugs are triggered by compound MRs. *This demonstrates that generating complex query statements through fusing or partitioning different graph-matching patterns can better stress the GDBs.* Out of these 30 bugs, 25 are detected using *pattern fusion MR* and five are *patterns partitioning MR*. Since GDBMeter has already used query partitioning to test the older versions of GDBs in our dataset for a long time, many related bugs have been found and fixed on the latest versions. Therefore, it is reasonable for Gamera to find only five such bugs. Among the detected bugs in this category, most of them are due to the omission of nodes or edges in the process of fusing patterns or decoupling patterns, resulting in incorrect result sets. There are also cases where nodes or edges are redundantly counted inside

Table 6: Types of the bugs found by Gamera and existing tools on latest version GDBs. Logic refers to logic bugs. Crash refers to crash bugs that can cause the GDB server to exit abnormally. Error refers to unexpected internal error bugs.

GDB	Gamera			Grand		GDsmith		GDBMeter
	Logic	Crash	Error	Crash	Error	Crash	Error	Logic
Neo4j	5	4	1	-	-	3	1	2
RedisGraph	6	1	2	-	-	1	2	1
OrientDB	3	0	0	0	0	-	-	1
JanusGraph	3	0	2	0	2	-	-	0
HugeGraph	2	3	2	2	1	-	-	1
TinkerGraph	1	1	2	1	2	-	-	0
ArcadeDB	1	0	0	0	0	-	-	0
Total	21	9	9	3	5	4	3	5

the results. The causes of these bugs include errors in numerical and value calculations, or design flaws in some query syntaxes, *etc.*

As for the dynamic MRs, four logic bugs were detected. These bugs include path retrieval errors after adding nodes and edges, as well as unexpected exceptions thrown when deleting the data. Though GDBs mostly well support the graph dynamic features such as various APIs for updating graph data, bugs still remain in the commonly-used APIs. Gamera could use its dynamic MRs to expose such bugs effectively. Specially, the data update and irrelevant data manipulation MR did not trigger any bug. Despite the fact that we constantly update graph data such as properties, and increase or reduce irrelevant graph data in large quantities, most GDBs can still function normally, and guarantee the correctness of their large-scale data manipulation.

Bug Types. We show in Table 6 the different types of bugs found by Gamera. *Though Gamera aims to find logic bugs, it has also detected crashes and errors.* Overall, there are 21 logic bugs. These bugs are distributed in every GDB, demonstrating the significance and necessity of detecting this type of vulnerability. We summarize five primary root causes that trigger the logic bugs. First, GDBs have erroneous support for path retrieval functionalities. Some exact path length queries or multi-hop queries do not follow the actual path length. Second, wrong numerical calculations in GDBs lead to inaccurate results. The bugs are caused by incorrect results when calculating Not a Number (NaN) values, $infinity$, and $strings$, *etc.* Third, some logic bugs are due to the incorrect implementation of the Gremlin APIs (*e.g.*, *outside()*, *gte()*, and *lte()*, *etc.*). Fourth, other logic bugs are caused by design flaws when querying for the intersection of patterns using *union* syntax or mid-traversal $E()$, *etc.* Last, there are also many logic bugs that come from erroneous omissions or redundant calculations of nodes or edges.

Gamera also found nine crash bugs and nine error bugs. Specifically, crash bugs are the type of bugs that cause the GDB server to exit while executing the queries. Error bugs mean that they would cause unexpected internal errors, but the GDBs can continue to process the subsequent queries. Crash and error bugs—though they might not be related to logic handling of GDBs—are important side-products because they can also cause severe consequences. For instance, when such crashes or errors occur in production GDBs, they would cause denial-of-service and interfere with all active users. Through manual analysis of these crash and error bugs, we further found that the root causes of these bugs can be traced back to some single queries. We can reproduce these bugs by executing only those single queries, without the need to apply a series of other MR queries.

For the remaining unconfirmed bugs, we are still actively communicating with the developers to help them confirm and fix the bugs. There are several reasons why they have not been confirmed in a timely manner. First, for bugs where path retrieval and matching nodes return incorrect results, although there have been some discussions, the developers have not yet located the root causes, hence unable to confirm these bugs. Second, for bugs in JanusGraph and HugeGraph, the developers suspect that they are due to the embedded TinkerPop server and have not yet confirmed whether the bugs are caused by their own database design. Third, developers are still determining if some bugs originate from the same root cause and try to deduplicate them. Moreover, we are still awaiting further responses from the developers regarding some other bugs.

6.3 Comparison with Existing Works

We further compare Gamera to related GDB benchmarking tools. As we have discussed in §2.3, Grand [47], GDsmith [32], and GDBMeter [34] are three closely related works for benchmarking GDBs. Grand and GDsmith test GDBs by leveraging differential testing, and GDBMeter is the only tool that utilizes metamorphic testing. These three works do not support or implement any oracles of Gamera, except that GDBMeter partially supports a simple version of the pattern partitioning oracle. We are able to include all of these three works into our comparison. In particular, they are all open-sourced [6–8]. We reuse the same set of GDBs in §6.1 for comparison. Specifically, we followed the identical experiment configurations and running procedures as described in their artifacts, and obtained the results consistent with their reports in the respective papers. These three experiments were all completed within 48 hours. The comparison results are shown in Table 6.

Comparison with Grand. We tried our best to conduct a fair comparison with Grand. Grand is designed for benchmarking Gremlin-based GDBs. We thus applied it on the five Gremlin-based GDBs for 10 rounds, each round generating 1,000 queries, using the same evaluation setup in §6.1. We then manually analyzed the bug reports to confirm true-positive bugs.

Gamera significantly outperforms Grand by identifying more bugs. Out of 10,000 queries, Grand reported a total of 1,420 potential bugs. We then manually analyzed Grand's bug reports to identify the true-positive bugs. Our analysis revealed that Grand could detect eight crash and error bugs in total, spanning all five Gremlin-based GDBs. It did not detect any logic bug, which demonstrates the need for our graph-aware MRs to reveal logic bugs. *These eight crash and error bugs were all successfully detected by Gamera.*

Gamera also beats Grand in terms of false-positive rate. The results also showed that Grand reported a large number of false positives (1,412 out of 10,000). The authors of Grand mentioned in their GitHub issue[2] that there are generally two reasons for the large false positive alarms: (1) the bugs in their tool such as wrong inputs or unsupported language syntaxes, and (2) Grand would classify the same exception between different GDBs as distinct ones and wrongly report bugs due to the different output string representations across GDBs. Gamera, on the other hand, has *no false*

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

positives since it is a metamorphic testing approach. The reasons why Gamera performs better than Grand are due to the limitations of Grand's differential oracle and its failure to consider the effective graph-aware MRs.

Comparison with GDsmith. We also tried to conduct a fair comparison with GDsmith. GDsmith is designed for benchmarking Cypher-based GDBs. We applied it on the two Cypher-based GDBs (*i.e.* Neo4j and RedisGraph) that Gamera currently supports. Similarly, we followed the same evaluation setup in §6.1 and use the same number of testing queries and testing rounds. We chose the differential oracle in GDsmith's artifact for testing. GDsmith reported a large number of discrepancies in its differential testing. Although we tried to manually eliminate the false positives and duplicated cases, without the help of GDsmith's authors or developers, we cannot give the exact number of true-positive bugs at the time of writing.

We finally decided to inspect whether GDsmith can detect those bugs that Gamera has already detected. The result showed that *Gamera presents better performances in the ability to detect logic bugs.* GDsmith could detect a total of seven crash and error bugs detected by Gamera, but could not detect any logic bugs that Gamera identified. Since GDsmith and Gamera have different testing oracles, they have different advantages and can complement each other. After examining GDsmith's bug reports, our analysis demonstrates that GDsmith cannot detect any logic bugs related to the graph native structures, since GDsmith does not support graph-aware MRs. For example, 11 bugs detected by Gamera in Neo4j and RedisGraph require the utilization of graph-aware MRs as testing oracles; GDsmith thus could not find any of them. However, Gamera cannot detect some bugs identified by GDsmith such as the bugs caused by the different support of Cypher standards in different GDBs, *etc.*, because GDsmith adopts a more sophisticated statement generation and mutation strategy. In the future, we will consider integrating both techniques to achieve better performance.

Comparison with GDBMeter. *Overall, GDBMeter could find only a small subset of bugs that Gamera could identify.* GDBMeter detected five logic bugs, which were *all successfully identified by Gamera*. These five bugs are mainly caused by miscalculations of the values, or design flaws of Gremlin APIs (*e.g.*, *inside()*, *outside()*, *etc.*). In fact, GDBMeter previously detected a total of 40 bugs on older versions of the tested GDBs. Since most of these types of bugs have been fixed in newer versions of GDBs, thus it is reasonable that we can hardly detect new bugs based on this oracle in the latest versions. The rest 34 bugs could not be detected by GDBMeter because triggering them requires support for advanced graph query language syntaxes (*e.g.*, path traversals, *union* clauses, *etc.*) and our graph-aware MRs.

6.4 Case Study

In this section, we showcase some interesting bugs Gamera found. We mainly show some bugs related to the graph native structures. We simplify some test cases for brevity.

K-hop Nodes Relationship Bug in RedisGraph. Listing 2 shows a case that produces false results that violate the *k-hop neighbour MR*. We construct the following scenario. The first query finds all the distinct *outgoing* 2-hop nodes of the node with id 1, which returns a result set containing that a node with id 2. The second query finds all the distinct *incoming* 2-hop nodes of the node with id 2. Obviously, the node with id 1 should exist in the result set of the second query. However, our testing found that RedisGraph omitted this node from the result set, leading to a logic bug. The developers confirmed this bug and fixed it timely. They admitted that the bug was due to an error that the multi-hop traversal in RedisGraph does not respect the actual path length.

```
1 MATCH path=(a)-[*2..2]->(b) WHERE (a<>b) AND (ID(a)=1)
      RETURN DISTINCT b
2 MATCH path=(a)<-[*2..2]-(b) WHERE (a<>b) AND (ID(a)=2)
      RETURN DISTINCT b
```

Listing 2: A k-hop node relationship bug in RedisGraph.

Union Fusion Bug in JanusGraph. Listing 3 shows one of the test cases about the logic bug when we fuse two patterns with the union relationship. In lines 1-2, we first randomly generate two queries, and then Gamera would calculate the union set of their results. The third query in line 3 utilizes *union*() syntax to fuse two queries matching two diverse patterns into a new one. However, its result is different from the former calculated union set. This discrepancy is due to the flaw in the design of *union*() in JanusGraph.

```
1 g.V().has('vp3',0.096).and(__.values('vp3'))
2 g.V().has('vl1', 'vp0',neq(18))
3 g.V().union(__.has('vp3',0.096).and(__.values('vp3')),
      __.has('vl1', 'vp0',neq(18))).dedup()
```

Listing 3: Incorrect result when merging queries in JanusGraph.

RedisGraph Incorrect Result after Graph Data Mutation. Listing 4 shows another logic bug in RedisGraph path search after mutating the graph data. We first count the number of paths from node one with id 3 and node two with id 1 (line 1). Then we create a new redundant node three and add a directed edge from node two to node three (lines 3-4). The expected result of line 5 is that the number of paths between node one and node three should be the same as the result of line 1. However, in the case of complex graphs, RedisGraph omits some paths and returns inaccurate results. This is because of the immature design of RedisGraph in terms of the path functionalities.

```
1 MATCH path=(a)-[*]->(b) WHERE ID(a)=3 AND ID(b)=1
      RETURN COUNT(path)
2 // -> 2
3 CREATE (n:new {key: 'val'}) RETURN ID(n)
4 MATCH (a),(b) WHERE ID(a)=1 AND ID(b)=104 CREATE (a)
      -[:e]->(b)
5 MATCH path=(a)-[*]->(b) WHERE ID(a)=3 AND ID(b)=104
      RETURN COUNT(path)
6 // -> 1 != 2
```

Listing 4: Inaccurate results after adding redundant nodes and edges.

Erroneous Value Handling in HugeGraph. In Listing 5, we show another two examples of logic bugs related to the incorrect value calculation. In HugeGraph, the Gremlin APIs (*e.g.*, *gte*(), *lt*(), *etc.*) do not correctly support some value calculations such as strings. We found these bugs when we used *pattern fusion MR* for testing. These types of values will be converted into floating-point numbers before performing operations. Thus the wrong conversions would

filter out the wrong results in line 1. Similarly, the Gremlin API *outside*() fails to properly handle the case of parameters with the same value, and would finally return incorrect results. HugeGraph developers confirmed these bugs and promised to fix them in the near future.

```
1 g.V().has('vp0',not(gte('iUn2s').or(lt('73'))))
2 g.V().has('vl0','vp1',outside(false,false)).has('vp1')
```

Listing 5: An erroneous value handling bug in Hugegraph.

7 DISCUSSION

Portability and Extensibility. GAMERA is a black-box MT approach tailored for GDBs using Cypher and Gremlin. We have benchmarked several commercial GDBs (*e.g.*, Neo4j, OrientDB, TinkerGraph). GAMERA can be directly applied to other GDBs using the query languages. For instance, GAMERA can be integrated into the GDB benchmarking process of the Apache TinkerPop community, which includes other GDBs. There are also many other research prototype GDBs such as GraphflowDB [35], LiveGraph [48], Grasper [24], and GTran [25], *etc.* GAMERA can be extended to support them with moderate engineering effort. Besides, the idea of designing MRs based on the graph native structures is generic. It can be leveraged to implement testing oracles for GDBs using different query languages beyond Cypher and Gremlin, such as the GSQL and AQL languages, *etc.* We open-source GAMERA and plan to contribute to the GDB community to help build precise and robust GDBs in the long term.

Approach Comprehensiveness and Completeness. We believe our approach is comprehensive and exhaustive as we have covered the graph-aware MRs related to the most commonly-used GDB operations and our compound-level MR could gradually generate complex ones from the elementary ones. Since there are always new feature updates to the GDBs as well as other corner cases, we admit that we cannot engineer all the graph-aware MRs, limited by our manpower. Nevertheless, we have implemented GAMERA in a highly extensible manner. Other researchers can add new testing oracles incorporating the ideas of elementary, compound, and dynamic MRs atop our open-source tool in the future.

Besides, as a common issue of dynamic bug detection approaches (including both DT and MT), there is always no guarantee that all logic bugs can be found. This is because the random nature of dynamic testing in terms of input generation and mutation makes it very challenging to achieve complete code and program state coverage. As a result, some potential logic bugs could remain undetected. Extending the experiment time might improve the comprehensiveness of the testing.

Limitations. GAMERA might generate duplicate bug reports, because it would generate distinct test cases to trigger bugs with the same root cause like other black-box benchmarking approaches. Our analysis currently requires human efforts to simplify the test cases and deduplicate the reports. Automated bug deduplication for logic bugs remains an open research challenge [18], and we leave it as a future exploration direction. In addition, GAMERA's current support of the complex syntaxes related to graph native structures is incomplete. There are still syntaxes that it does not support, *e.g.*, graph algorithm operations (*e.g.*, *pageRank*() [9]), *etc.* We plan to supplement GAMERA with these syntax supports to further improve its effectiveness in the future.

8 RELATED WORK

DT for Database Systems. DT [38] is a commonly used approach to detect bugs in different domain systems. In relational database systems, one direction [31, 45] is to run queries with different relational databases and check the inconsistency of the results. Another direction [33, 46] is to run queries with different settings of one relational database, like versions or optimization levels. RAGS [45] first leverages DT to detect logic bugs in relational database systems. APOLLO [33] uses DT to find performance regression bugs. As for graph database systems, GDsmith [32] and Grand [47] are the first work to use DT to detect bugs in Cypher-based GDBs and Gremlin-based GDBs, respectively. According to the number of bugs eventually confirmed in different GDBs, both of these works prove to be effective bug detection tools. However, as discussed in §2.3, they share some fundamental limitations of DT. This paper leverages MT to tackle the oracle problem and performs a more comprehensive benchmarking of every single graph database.

MT for Database Systems. MT [26] is first proposed to alleviate the absence of testing oracles. The main idea of this approach is to design MRs and validate the outputs. SQLancer [18] is designed to detect logic bugs in relational database systems. It integrates several novel MT approaches. NoREC [42] compares the execution results of a given optimized query with its non-optimized version, to detect bugs related to optimization. TLP [41] partitions a query into three sub-queries, and detects logic bugs by comparing the combination result of three sub-queries with the original result. GDBMeter [34] also leverages the idea of TLP to detect logic bugs in GDBs. However, all the above methods are designed for database systems using query languages such as those SQL-like or other graph query languages. These methods do not consider the graph native structures and correspondingly propose a set of graph-aware MRs. We have thoroughly analyzed these techniques in §2.3.

9 CONCLUSION

Logic bugs in GDBs related to graph native structures are underexplored. In this work, we proposed GAMERA, an effective and automated metamorphic testing approach to detecting logic bugs in GDBs. We developed three classes of novel graph-aware MRs—elementary MRs, compound MRs and dynamic MRs—for generating high-quality tests to cover diverse and complex GDB operations. Our evaluation of GAMERA on seven widely-used Cypher-based and Gremlin-based GDBs found 39 bugs, 15 of which have been confirmed by the developers. GAMERA also significantly outperformed previous works by detecting all bugs they could find. Our research demonstrates the importance of employing graph-aware MRs for detecting logic bugs in GDBs.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their helpful suggestions and comments. The work described in this paper was partly supported by a grant from the Research Grants Council of the Hong Kong SAR, China (Project No.: CUHK 14209323).

REFERENCES

[1] 2015. Count Lines of Code. https://github.com/AlDanial/cloc.

[2] 2023. AQL (ArangoDB Query Language). https://www.arangodb.com/docs/stable/aql/.

[3] 2023. ArangoDB. https://www.arangodb.com/.

[4] 2023. Cypher Manual. https://neo4j.com/docs/cypher-manual/current/introduction/.

[5] 2023. DB-Engines Ranking of Graph DBMS. https://db-engines.com/en/ranking/graph+dbms.

[6] 2023. GDBMeter artifact. https://github.com/gdbmeter/gdbmeter.

[7] 2023. GDsmith artifact. https://github.com/ddaa2000/GDsmith.

[8] 2023. Grand artifact. https://github.com/tcse-iscas/Grand.

[9] 2023. Graph Algorithms. https://neo4j.com/docs/graph-data-science/current/algorithms/.

[10] 2023. Gremlin Manual. https://tinkerpop.apache.org/docs/3.6.2/reference/.

[11] 2023. HugeGraph. https://hugegraph.apache.org/.

[12] 2023. JanusGraph. https://janusgraph.org/.

[13] 2023. Neo4j. https://neo4j.com/.

[14] 2023. The Next Generation Multi-Model Database Supporting Graphs, Key/Value, Documents and Time-Series. https://arcadedb.com/.

[15] 2023. OrientDB. https://orientdb.org/.

[16] 2023. Path Finding. https://neo4j.com/docs/graph-data-science/current/algorithms/pathfinding/.

[17] 2023. RedisGraph. https://redis.io/docs/stack/graph/.

[18] 2023. SQLancer. https://github.com/sqlancer/sqlancer.

[19] 2023. TingerGraph. https://www.tigergraph.com/.

[20] 2023. TinkerGraph. https://github.com/tinkerpop/blueprints/wiki/tinkergraph.

[21] 2023. TinkerPop. https://tinkerpop.apache.org/.

[22] 2023. Traversal Recipes. https://tinkerpop.apache.org/docs/current/recipes/#_traversal_recipes.

[23] Marcelo Arenas, Claudio Gutiérrez, 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*. 2821–2828.

[24] Hongzhi Chen, Changji Li, Juncheng Fang, Chenghuan Huang, James Cheng, Jian Zhang, Yifan Hou, and Xiao Yan. 2019. Grasper: A high performance distributed system for OLAP on property graphs. In *Proceedings of the ACM Symposium on Cloud Computing*. 87–100.

[25] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jian Zhang. 2022. G-tran: a high performance distributed graph database with a decentralized architecture. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2545–2558.

[26] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases*. Technical Report. HKUST-CS98-01, Department of Computer Science, Hong Kong.

[27] Forrester Consulting. 2021. The Total Economic Impact™ of the Neo4j Graph Data Platform. https://neo4j.com/whitepapers/forrester-total-economic-impact/.

[28] Alin Deutsch. 2018. Querying Graph Databases with the GSQL Query Language.. In *SBBD*. 313.

[29] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*. Melbourne, Victoria, Australia.

[30] 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*. 1433–1445.

[31] Bogdan Ghit, Nicolas Poggi, Josh Rosen, Reynold Xin, and Peter Boncz. 2020. SparkFuzz: searching correctness regressions in modern query engines. In *Proceedings of the workshop on Testing Database Systems*. 1–6.

[32] Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, Wenpin Jiao, and Tao Xie. 2023. GDsmith: Detecting bugs in Cypher graph database engines. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[33] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment* 13, 1 (2019), 57–70.

[34] M Kamm, M Rigger, C Zhang, and Z Su. 2023. Testing graph database engines via query partitioning. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[35] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *Proceedings of the 2017 ACM SIGMOD/PODS Conference*. Chicago, Illinois, USA.

[36] Baozhu Liu, Xin Wang, Pengkai Liu, Sizhuo Li, Qiang Fu, and Yunpeng Chai. 2021. UniKG: A unified interoperable knowledge graph database system. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2681–2684.

[37] Pingchuan Ma and Shuai Wang. 2021. MT-teql: evaluating and augmenting neural NLIDB on real-world linguistic and schema variations. *Proceedings of the VLDB Endowment* 15, 3 (2021), 569–582.

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

[39] Neo4j. 2015. Introducing the new Cypher Query Optimizer. https://neo4j.com/blog/introducing-new-cypher-query-optimizer/.

[40] Yuxiang Ren, Hao Zhu, Jiawei Zhang, Peng Dai, and Liefeng Bo. 2021. Ensemfdet: An ensemble approach to fraud detection based on bipartite graph. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2039–2044.

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

[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] Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. 1–10.

[44] Chandan Sharma and Roopak Sinha. 2019. A schema-first formalism for labeled property graph databases: Enabling structured data loading and analytics. In *Proceedings of the 6th ieee/acm international conference on big data computing, applications and technologies*. 71–80.

[45] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. Citeseer, 618–622.

[46] Jiaqi Yan, Qiuye Jin, Shrainik Jain, Stratis D Viglas, and Allison Lee. 2018. Snowtrail: Testing with production queries on a cloud database. In *Proceedings of the Workshop on Testing Database Systems*. 1–6.

[47] 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.

[48] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: a transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1020–1034.