Testing Graph Database Systems with Graph-State Persistence Oracle

Shuang Liu
Key Laboratory of Data Engineering and Knowledge Engineering (MOE), School of Information, Renmin University of China
Beijing, China
Shuang.Liu@ruc.edu.cn

Junhao Lan
College of Intelligence and Computing, Tianjin University
Tianjin, China
lanjunhao@tju.edu.cn

Xiaoning Du
Monash University
Monash, Australia
Xiaoning.Du@monash.edu

Jiyuan Li
College of Intelligence and Computing, Tianjin University
Tianjin, China
lijiyuan@tju.edu.cn

Wei Lu*
Key Laboratory of Data Engineering and Knowledge Engineering (MOE), School of Information, Renmin University of China
Beijing, China
lu-wei@ruc.edu.cn

Jiajun Jiang
College of Intelligence and Computing, Tianjin University
Tianjin, China
jiangjiajun@tju.edu.cn

Xiaoyong Du
Key Laboratory of Data Engineering and Knowledge Engineering (MOE), School of Information, Renmin University of China
Beijing, China
duyong@ruc.edu.cn

Abstract

Graph Database Management Systems (GDBMSs) store data in a graph format, facilitating rapid querying of nodes and relationships. This structure is particularly advantageous for applications like social networks and recommendation systems, which often involve frequent writing operations—such as adding new nodes, creating relationships, or modifying existing data—that potentially introduce bugs. However, existing GDBMS testing approaches tend to overlook these writing functionalities, failing to detect bugs arising from such operations. In this paper we present GraspDB, the first metamorphic testing approach specifically designed to identify bugs related to writing operations in graph database systems. GraspDB employs the Graph-State Persistence oracle, which is based on the Labeled Property Graph Isomorphism (LPG-Isomorphism) and Labeled Property Subgraph Isomorphism (LPSG-Isomorphism) relations. We also develop three classes of mutation rules aimed at engaging more diverse writing-related code logic. GraspDB has successfully detected 77 unique, previously unknown bugs across four popular open source graph database engines, among which 58 bugs are confirmed by developers, 43 bugs have been fixed and 31 are related to writing operations.

*Wei Lu is the corresponding author.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**.

Keywords

Database Testing, Graph Databases, Metamorphic Testing

ACM Reference Format:
Shuang Liu, Junhao Lan, Xiaoning Du, Jiyuan Li, Wei Lu, Jiajun Jiang, and Xiaoyong Du. 2024. Testing Graph Database Systems with Graph-State Persistence Oracle. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3680311

1 Introduction

In recent years, Graph Database Management Systems (GDBMSs) - [33] have been widely used in various applications where data is represented by vertices and edges, such as social networks [15, 38], knowledge graphs [13], and recommendation systems [12, 28, 40]. GDBMSs offer efficient data storage and querying capabilities, enabling these applications to fulfill user requests with high throughput, thereby becoming a cornerstone of application driven by graph data. According to industry reports, the global market size for graph databases is projected to reach $2.9 billion in 2023 [5], underscoring the growing demand for such systems. Among the various GDBMS implementations, Neo4j [10] stands out as the most popular and widely adopted graph data platform, trusted by over 1,000 organizations, including industry leaders such as Adobe, eBay and UBS [36]

Similar to other databases management systems (DBMSs), GDBM -Ss accept user queries as input and return the corresponding query results. These user queries convey requests to store, manipulate, or retrieve data from the databases. To ensure better reliability and efficiency, DBMSs demand sound and optimized implementations of transaction management, indexing and storage management, query optimization, and concurrency control. The combination of these factors, coupled with the inherent complexity of database systems, renders their implementation susceptible to bugs. Recognizing this challenge, the software testing community has devoted increasing attention to uncovering bugs in DBMS, with a particular focus on GDBMSs [20–22, 41].

In order to reveal the deeply buried logic bugs in GDBMSs, one of the biggest challenges lies in devising powerful test oracles. As a countermeasure, differential testing [20, 41] and metamorphic testing [21, 22, 25, 42] play an crucial role in existing works on GDBMS testing. The insight behind differential testing is that different GDBMS implementations should return the same results when dealing with identical user requests. However, different GDBMSs may adopt different graph query languages, such as Cypher [18] and Gremlin [34], which limits the application of differential testing approaches to a specific group of implementations that support the same query language. GDSmith [20] and Grand [41] are the state-of-the-art differential testing approaches for GDBMSs, respectively supporting Cypher and Gremlin. Another factor that influences the effectiveness of differential testing is the absence of standard specifications for graph query languages, leading to different implementation choices even for the same query language. Failing to carefully verify whether two implementations adhere to the same specification also result in false alarms [21].

In contrast, metamorphic testing approaches offer more precise oracles for testing GDBMSs, and a body of works surge in this line, including GDBMeter [22], GraphGenie [21], and GRev [25]. They have particularly focused on the data retrieval functionality of GDBMSs and designed metamorphic relations that capture whether and how the query results are affected when particular transformations are made to the matching conditions. For example, GRev proposes representing the matching patterns with Abstract Syntax Graph (ASG) and developing an algorithm to extract equivalent matching conditions from the graph, which describes exactly the original pattern but with a different set of conditions, thereby rewriting the queries. If the query results differ, a bug is caught. Additionally, GAMERA [42] proposes metamorphic relations based on the intrinsic properties of graphs, including symmetric relations such as connectivity, and inverse relations such as ancestor and descendant. We can observe that existing works have exclusively focused on the pattern-matching and data-retrieving functionalities of GDBMSs, leaving their capabilities to writing operations, e.g., write, update, and delete data, rarely examined. Hence, **we make the first endeavor to test how well the GDBMSs handle queries with writing operations.**

```
WITH 1 AS a WHERE NULL CREATE (a);
--Expected Behavior: 0 node created
--Actual Behavior: 1 node created
```

Figure 1: A query with CREATE generated by our approach that triggers a non-crashing bug in RedisGraph.

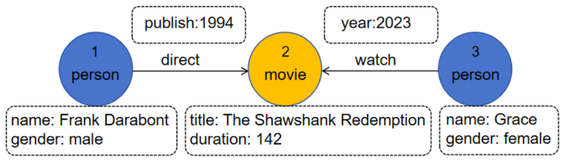
Bugs may or may not cause crashes. A reliable GDBMS implementation shall never crash for any user query. This also applies to how it processes queries with writing operations.

Figure 1 shows a query with CREATE operation, which is generated by our approach and triggers such a non-crashing bug in RedisGraph. The bug was detected using the query pair in Figure 1 and its mutated query "WITH 1 AS a WHERE NULL WITH * CREATE (a)" (via applying mutation Rule14 in Table 2), with our Graph-State Persistent Oracle. There are three parts in the query. `WITH 1 AS a` is a projecting clause, which passes a record with variable `a` with value `1` to the next part. `WHERE NULL` filters the results from the `WITH` clause. In this case, the condition `NULL` evaluates to false, and no record is eligible for the following clause. Thus, the `CREATE (a)` clause will not be ignited, and no nodes shall be created. However, when RedisGraph executes this query, it creates one node. The bug is in some sense "silent" as it does not show explicit error messages and only checking on the database status can expose the bug. Note that this bug can not be found by existing approaches for two reasons. Firstly, existing approaches do not generate queries with update-related operations and thus cannot produce queries with CREATE. Secondly, the oracle hired by existing approaches does not specify properties on the graph state and thus cannot capture the bug.

To address the challenges, we propose an enhanced graph-state persistent oracle to detect bugs from writing operations, using the Labeled Property Graph Isomorphism (LPG-Isomorphism) and Labeled Property Subgraph Isomorphism (LPSG-Isomorphism) metamorphic relations. We propose three classes of mutation rules designed to increase the likelihood of triggering bugs by adding writing clauses or modifying existing queries with writing operations, thereby engaging more diverse writing-related code logic. These mutation rules are also designed guided by the LPG-Isomorphism and LPSG-Isomorphism relations to ensure the execution correctness of the mutated query and the base query pairs verifiable with these isomorphism relations. We conduct experiments with 4 popular GDBMSs and detect 77 bugs, among which 58 bugs are confirmed and 43 have been fixed. There are 31 confirmed bugs related to writing operations, and thus cannot be detected by all existing approaches that test GDBMS.

In summary, we make the following contributions:

- We propose the first metamorphic testing approach for detecting writing-related bugs in GDBMSs by employing the Graph-State Persistence oracle based on the Labeled Property Graph Isomorphism (LPG-Isomorphism) and Labeled Property Subgraph Isomorphism (LPSG-Isomorphism) relations. We propose three classes of mutation rules to engage more diverse writing-related code logic.
- We conduct experiments on four commercial GDBMSs. Our approach detected 77 previous unknown bugs. 58 of them



(a) A labeled property graph example

```
MATCH (a:person)-[:direct]->(:movie)<-[:watch]-(b:{gender:'female'})
WHERE a.name = 'Frank Darabont' RETURN b.name AS n
```

(b) A Cypher query example

Figure 2: Labeled property graph and Cypher query examples.

Table 1: Clauses of Cypher Language [9]

Reading clauses := MATCH|OPTIONAL MATCH	
Writing clauses := CREATE|MERGE|FOREACH|SET|REMOVE|(DETACH) DELETE	
Projecting clauses:= WITH|UNWIND|RETURN	
Subquery clauses := CALL	

have been confirmed by the developers, and 43 have been fixed. Among them, there are 11 logic bugs, 32 errors, 13 crashes and 2 inconsistencies with Cypher documentation. Additionally, 31 bugs cannot be triggered without database writing operations.

- We have implemented our method as a practical tool called GraspDB and the source code is available at https://doi.org/10.5281/zenodo.12670528.

2 Preliminary

2.1 Labeled Property Graph and Cypher

In this section, we provide preliminaries on the labeled property graph model and the Cypher language [18].

Labeled property graph model (LPG) is a data model for representing and storing data in GDBMS. Neo4j, MemGraph, RedisGraph and AgensGraph are examples of GDBMSs that use the labeled property graph model. A labeled property graph has a set of nodes and relationships (directed edges that connect nodes). Each node and edge (relationship[1]) can have a set of properties, which are key-value pairs and are usually specified using JavaScript Object Notation (JSON). Each node has a set of unique labels, which are tokens that describe the type of the node. Each edge has one label, indicating the edge type. Figure 2(a) shows an example labeled property graph. The LPG contains three nodes, two labeled with `person` and one labeled with `movie`, and two edges, labeled with `direct` and `watch`, respectively. The `person` nodes have two properties, i.e., name and gender. The `movie` node has two properties, i.e., title and duration. The `direct` and `watch` edges are associated with properties publish and year, respectively.

Cypher [18] is an declarative programming language originally developed for the Neo4j graph database [10]. Cypher is easy to read and write due to its declarative nature, and it is known for its expressive and efficient way to handle patterns within graphs, making it well-suited for complex queries. Pattern matching is conducted to retrieve subgraphs from a property graph in Cypher. In each Cypher query, clauses are chained together and executed sequentially. Each Cypher clause takes the property graph and the intermediate results of the previous clause as input, and output the intermediate results to the next clause. Table 1 lists the four types of clauses defined in Cypher, including reading clauses, writing clauses, projecting clauses and subquery clauses. Figure 2(b) shows an example Cypher query that aims to find the female viewers of movies directed by Frank Darabont. In this query, `(a:person)`, `(:movie)` and `(b{gender:'female'})` are node patterns, `-[:direct]->` and `<-[:watch]-` are relationship patterns, `a.name = 'Frank Darabont'` is an expression. The name of the field can be renamed by using `AS`.

Unlike the Structured Query Language (SQL) [1], there is no standard specification for graph query languages and thus there are various query languages for GDBMS [27, 35]. Cypher [27], originally contributed by Neo4j [10], is widely recognized with the wide adoption of Neo4j, and it is used by over 10 other popular databases including RedisGraph [11] and Memgraph [8]. Cypher is regarded as the most widely adopted, fully-specified, and open query language for property graph database engines [17, 18]. Some graph databases that natively support other graph query languages (e.g., Gremlin [34]) are also compatible with Cypher queries via translation tools (e.g., Cypher for Gremlin [3]).

2.2 Graph Isomorphism

We provide preliminaries of definitions on the isomorphism of graphs, which serves as the basis for our graph-state persistence metamorphic relations, in this section.

Definition 2.1 (Graph Isomorphism [37]). Given two graphs G=(V, E) and G'=(V', E'), where V and V' are node sets and E and E' are edge sets. G and G' are called isomorphic if there exist an edge-preserving bijection mapping $f : V \longrightarrow V'$, such that $\forall u, v \in V$, $(u, v) \in E$ iff $(f(u), f(v)) \in E'$. We denote G $\simeq$ G' if G and G' are isomorphic graphs.

Definition 2.2 (Labeled Graph [23]). A labeled graph is defined as G=(V, E, L, l_V, l_E), where V is the node set, E is the edge set, L is the set of node labels and edge labels. $l_V : V \longrightarrow \mathbb{P}(L)$is the mapping from node to a set of labels ($\mathbb{P}(L)$ is the power set of L), and $l_E : E \longrightarrow L$ is the mapping from edge to labels.

Definition 2.3 (Labeled Graph Isomorphism [19]). Given two labeled graphs G=(V, E, L, l_V, l_E) and G'=(V', E', L', l'_V, l'_E), G and G' are isomorphic if (1) there exists a bijective function $f : V \longrightarrow V'$, such that $\forall u, v \in V$, $(u, v) \in E$ iff $(f(u), f(v)) \in E'$; (2) $\forall u \in V$, $l_V(u) = l'_V(f(u))$; (3) $\forall u, v \in V$, $l_E(u, v) = l'_E(f(u), f(v))$. We denote G $\simeq_{LG}$ G' if G and G' are isomorphic labeled graphs.

Definition 2.4 (Subgraph [37]). Given two graphs G=(V, E), G'=(V', E'), where V and V' are node sets and E and E' are edge sets, G is a subgraph of G' if $V \subseteq V'$ and $E = E' \cap (V \times V)$. We denote $G \subseteq G'$ if G is a subgraph of G'.

Definition 2.5 (Subgraph Isomorphism [37]). Graph G=(V, E) is subgraph isomorphic to graph G'=(V', E') if $\exists S$ such that $S \subseteq G'$ and $S \simeq G$. We denote G $\sim$ G' if G is subgraph isomorphic to G'.

[1]We use edge and relationship interleaving in the following.

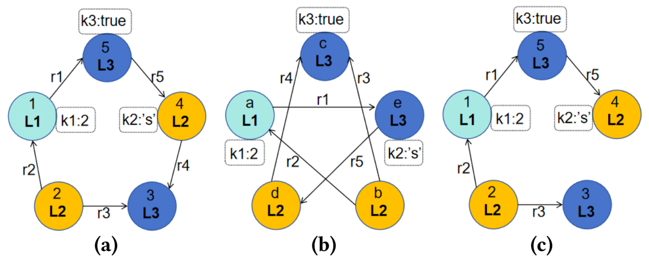


Figure 3: Labeled Property Graph Examples

Definition 2.6 (labeled Subgraph [32]). Given two labeled graphs G=(V, E, L, l_V, l_E) and G'=(V', E', L, l'_V, l'_E), G is a subgraph of G' if (1) there is an injective mapping f: $V \longrightarrow V'$ such that $\forall u, v \in V$, $(u, v) \in E \Rightarrow (f(u), f(v)) \in E'$; (2) $\forall u \in V$, $l_V(u) = l'_V(f(u))$; and (3) $\forall u, v \in V$, $l_E(u, v) = l'_E(f(u), f(v))$. We denote $G' \subseteq_{LG} G$ if G' is a subgraph of G.

Definition 2.7 (Labeled Subgraph Isomorphism [6]). Labeled Graph G=(V, E, L, l_V, l_E) is subgraph isomorphic to labeled graph G'=(V', E', L, l'_V, l'_E) if $\exists S$ such that $S \subseteq_{LG} G$ and $S \simeq_{LG} G'$. We denote G $\sim_{LG}$ G' if G is labeled subgraph isomorphic to G'.

3 Isomorphism Relations of Labeled Property Graph

Inspired by the graph isomorphism relations, we define isomorphism relations on labeled property graphs, which serve as the theory support of our oracle.

Definition 3.1 (Labeled Property Graph). A labeled property graph is defined as G=(V, E, L, P, l_V, l_E, f_V, f_E), where V, E are the set of nodes and edges; L is the set of labels for nodes and edges; P is the set of properties associated with nodes and edges. $l_V : V \longrightarrow \mathbb{P}(L)$ is the mapping from node to a set of labels ($\mathbb{P}(L)$ is the power set of L), and $l_E : E \longrightarrow L$ is the mapping from edge to labels. $f_V : V \longrightarrow \mathbb{P}(P)$ is the mapping from node to a set of node properties and $f_E : E \longrightarrow \mathbb{P}(P)$ is the mapping from edge to a set of edge properties.

Figure 3(a) is an example labeled property graph with 5 nodes and 5 edges. Each node is associated with a set of labels, e.g., {L1} for node 1 and a set of properties, e.g., {K1 : 2} for node 1. Each edge is associated with 1 unique label (or type), e.g., `r3` for edge `(2, 3)` and a set of properties, in this case an empty set for edge `(2, 3)`.

Definition 3.2 (Labeled Property Graph Isomorphism). Given two labeled property graph G=(V, E, L, P, l_V, l_E, f_V, f_E) and G'=(V', E', L', P', l'_V, l'_E, f'_V, f'_E), G and G' are isomorphic if (1) there exists a bijective mapping $f : V \longrightarrow V'$, such that $\forall u, v \in V$, $(u, v) \in E$ iff $(f(u), f(v)) \in E'$; (2) $\forall u \in V$, $l_V(u) = l'_V(f(u))$, $f_V(u) = f'_V(f(u))$; (3) $\forall u, v \in V$, $l_E(u, v) = l'_E(f(u), f(v))$, $f_E(u, v) = f'_E(f(u), f(v))$. We denote G $\simeq_{LPG}$ G' if G and G' are isomorphic labeled graphs.

Definition 3.3 (Labeled Property Subgraph Isomorphism). Given two labeled property graphs G=(V, E, L, P, l_V, l_E, f_V, f_E) and G'=(V', E', L', P', l'_V, l'_E, f'_V, f'_E), G is subgraph isomorphic to G' if (1) there is an injective mapping f: $V \longrightarrow V'$ such that $\forall u, v \in V$, $(u, v) \in E \Rightarrow (f(u), f(v)) \in E'$; (2) $\forall u \in V$, $l_V(u) = l'_V(f(u))$ and $f_V(u) = f'_V(f(u))$; (3) $\forall u, v \in V$, $l_E(u, v) = l'_E(f(u), f(v))$, $f_E(u, v) = f'_E(f(u), f(v))$. We denote G $\sim_{LPG}$ G' if G is labeled property subgraph isomorphic to G'.

A labeled property graph isomorphism is a bijective relation between two labeled property graphs, requesting two labeled property graphs to be structurally isomorphic, and the corresponding nodes and edges have identical labels and properties. Labeled property subgraph isomorphism defines an injective relation between two labeled property graphs, requesting two labeled property graphs to be structurally subgraph isomorphic, and the corresponding nodes and edges have identical labels and properties. The LPG in Figure 3(b) is isomorphic to that in Figure 3(a), and the LPG in Figure 3(c) is a subgraph isomorphic to that in Figure 3(a) according to our definitions. The two isomorphism relations on labeled property graphs serve as the theoretical basis for our oracle.

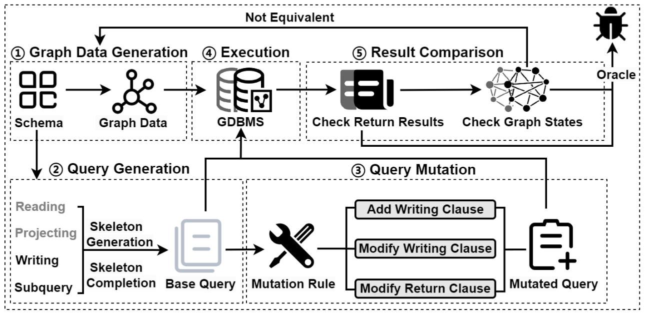


Figure 4: Overview of GraspDB

4 Approach

There are two main technical challenges for detecting bugs introduced by writing operations in graph databases. The first challenge is to precisely detect the bugs with low false positives. Since the bugs triggered by writing operations may modify the underlying graph database and the bug symptoms may not be reflected by query results, as illustrated in Figure 1 (none of existing graph database testing approaches is able to detect that kind of bugs). The second challenge is to generate test cases which can effectively trigger bugs arising from writing operations.

To address the first challenge, we propose the graph-state persistent oracle, which is based on two metamorphic relations, i.e., labeled property graph isomorphism (LPG-isomorphism) and labeled property subgraph isomorphism (LPSG-isomorphism). To address the second challenge, we have developed three categories of mutation rules, i.e., add writing clauses, modify writing clauses and modify return clauses, which are guided by the LPG-Isomorphism and LPSG-Isomorphism metamorphic relations. These mutation rules incorporate a variety of writing-related syntax features into the test cases, aiming to engage more writing-related code logic and consequently uncover more bugs. Additionally, we proactively generate test cases that incorporate writing clauses and sub-clauses. The writing clauses modify the graph and are likely to expose bugs, while the sub-clauses enhance the complexity of the test cases, increasing the likelihood of triggering bugs. We apply the mutation rules on base queries to obtain pairs of base and mutated

Table 2: Mutation Rules, with Colored ~~Deletion~~ and Addition

ID	Type	Oracle		Transformation	Example Query
		Graph	Result		
01	AWC	$\simeq_{LPG}$	=	Create nodes/edges and then delete	CREATE (a) CREATE p=()-[:T]→() DELETE p RETURN a
02	AWC	$\simeq_{LPG}$	=	Add a property and then delete	CREATE (a) SET a.k=1 REMOVE a.k RETURN a
03	AWC	$\simeq_{LPG}$	=	Remove a non-existent property	CREATE (a) REMOVE a.k RETURN a
04	AWC	$\simeq_{LPG}$	=	Delete a non-existent node/edge	CREATE (a) DELETE NULL RETURN a
05	AWC	$\sim_{LPG}$	=	Create nodes/edges	CREATE (a) CREATE ()-[:T]→() RETURN a
06	AWC	$\sim_{LPG}$	=	Delete existent nodes/edges	CREATE (a) DELETE a RETURN a
07	AWC	$\simeq_{LPG}$	=	Create path incrementally	CREATE ~~(a)-[:T]→(b)-[:T]→(c)~~(a)-[:T]→(b) CREATE (b)-[:T]→(c) RETURN a
08	AWC	$\simeq_{LPG}$	=	Delete path incrementally	CREATE p=()-[:T]→() ~~DELETE p~~FOREACH(f in nodes(p)|DETACH DELETE f)
09	MWC	$\simeq_{LPG}$	=	Add path variable	CREATE p=(a)-[:T]->() RETURN a
10	MWC	$\simeq_{LPG}$	=	Add node/edge variable	CREATE (a)-[:T]->(b) RETURN a
11	MWC	$\simeq_{LPG}$	=	Reverse the direction of the path	CREATE ~~(a)-[:T]→(b)~~(b)←[:T]-(a) RETURN a
12	MWC	$\simeq_{LPG}$	=	Wrap writing clause with FOREACH/CALL	FOREACH(f in [1]|CREATE p=(a)-[:T]→()) RETURN 1
13	MWC	$\simeq_{LPG}$	=	Add redundant UNWIND clause before writing clause	UNWIND [1] as l CREATE p=(a)-[:T]→() RETURN a
14	MWC	$\simeq_{LPG}$	=	Add redundant WITH clause before writing clause	WITH * CREATE p=(a)-[:T]→() RETURN a
15	MWC	$\simeq_{LPG}$	=	Add redundant OPTIONAL MATCH clause before writing clause	OPTIONAL MATCH ()-[:TYPE]-() CREATE p=(a)-[:T]→() RETURN a
16	MWC	$\sim_{LPG}$	NA	Increase the size of graph	CREATE (a)-[:T]→(c)-[:T]→(b) RETURN a
17	MWC	$\sim_{LPG}$	NA	Decrease the size of graph	CREATE (a)~~-[:T]→(c)~~ RETURN a
18	MRC	$\simeq_{LPG}$	$\subseteq$	Add records to return result	CREATE (a) RETURN 1 as n UNION RETURN 2 as n
19	MRC	$\simeq_{LPG}$	$\supseteq$	Return limited records	CREATE (a)-[:T]→(c) RETURN a LIMIT 0
20	MRC	$\simeq_{LPG}$	$\supseteq$	Return distinct records	CREATE (a)-[:T]→(c) RETURN DISTINCT a
21	MRC	$\simeq_{LPG}$	=	Change the order of return result	CREATE (a)-[:T]→(c) RETURN a ORDER BY a DESC
22	MRC	$\simeq_{LPG}$	=	Add a column to return result	CREATE (a)-[:T]→(c) RETURN a, c
23	MRC	$\simeq_{LPG}$	=	Wrap return result in a list	CREATE (c) RETURN [c][0]
24	MRC	$\simeq_{LPG}$	=	Wrap return result with reduce()	CREATE (c) RETURN reduce(a = c, b in [], a)

The mutated graph is LPSG-isomorphic to the base graph in the case of rule 6 and rule 17.
NA signifies that the relationship between the return results before and after mutation is unknown, and we do not compare the return results for this mutation rule.

queries, the execution correctness of which can be verified with the metamorphic relations.

The overview of GraspDB is shown in Figure 4, which consists five steps. The first two steps are graph database and query generation. GraspDB utilizes the graph database generation functionality of GDSmith [20] and improves it to generate base queries with all four types of clauses, in which enable generating writing clauses and subquery clauses could engage writing-related, complex code processing logic. Then, GraspDB mutates a base query based on three classes of mutation rules, i.e., add writing clause, modify writing clause, modify return clause, and obtains the follow-up mutated queries, such that the execution correctness of the base query and mutated query pairs can be verified with the metamorphic relations. In the fourth step, the pair of base query and mutated query is executed separately on two identically configured GDBMS instances, during which errors and crashes are directly detected. Finally, GraspDB detects logic bugs by comparing the graph database instances modified by the executed query pairs, as well as the return results (to reduce false negatives). Note that once there is an inconsistency in the two graph database instances, GraspDB will clear the two databases by deleting all nodes and go back to the first step, so as to prepare consistent execution environments for the next query pairs.

4.1 Query Generation

To detect bugs introduced by writing operations in GDBMSs, we need to generate initial graphs and semantically correct queries which contain update clauses. We improve GDSmith [20], a syntax-based GDBMS test case generation tool, for property graph and query generation by incorporating writing clauses and the corresponding patterns and expressions. Following the workflow of GDSmith, we conduct skeleton generation, pattern generation and expression generation, adhering strictly to grammar rules of Cypher. For skeleton generation, we refer to the syntax and semantics of Cypher and add skeletons related to writing clauses and subquery clauses. For pattern generation, we record the paths that are added to the initial graph due to writing clauses such as `CREATE` and `MERGE`, and proactively generate patterns based on those paths so as to improve the probability of retrieving the newly added nodes and edges. For expression generation, we extend GDSmith to add three types of expressions, i.e., function expressions (including function nesting), subquery expressions (e.g., EXISTS subquery and COUNT subquery) and some constant value expressions such as paths, lists, dictionaries. We also add some Cypher features, such as UNION keyword, CASE WHEN keyword, Pattern comprehension, List comprehension and Map projection, that GDsmith doesn't support, with the purpose of increasing diversity of the generated queries. We follow Cypher syntax and semantics, and the generation process of GDSmith to generate syntax and semantic correct base queries.

4.2 Mutation Rules

In this section, we provide the details of mutation rules corresponding to our metamorphic relations. In particular, we propose three classes of mutation rules, i.e., add writing clauses (AWC), modify writing clauses (MWC), and modify return clauses (MRC), based on the strategies that used. The mutation rules are guided by the metamorphic relations we defined in section 3, with Labeled Property Graph Isomorphism (LPGI) marked $\simeq_{LPG}$ and Labeled Property Subgraph Isomorphism (LPSI) marked $\sim_{LPG}$ in Table 2.

Add Writing Clause (AWC) Rules. This class of rules add writing clauses to an existing query, with the purpose of generating queries that trigger writing related functionalities and thus uncover writing-related bugs. The AWC rules are derived from both the LPG-isomorphism and LPSG-isomorphism relations, where they transform a based query into a mutated query that create an isomorphic labeled property graph or an isomorphic labeled property subgraph. For instance, rule 01 in Table 2 is an AWC mutation rule, which transforms the base query into a mutated query that create an isomorphic LPG with that created by the base query, by creating a path p and delete it immediately. Rule 05 transforms the base query into a mutate query that create a LPG G', which satisfies $G \subseteq_{LPSG} G'$, where G is the LPG created by the base query.

Modify Writing Clause (MWC) Rules. This class of rules modifies existing writing clauses, either directly by modifying the path/-pattern in a writing clause, or indirectly by adding clauses which potential affect the execution plan of an existing writing clause. The MWC rules are derived from both the LPG-isomorphism and LPSG-isomorphism relations. Rule 11 in Table 2 is a MWC rule which modifies the create clause in the base query to obtain a mutated query by reversing the direction of the created path. The mutated query creates an isomorphic LPG with that created by the base query. Rule 17 is a MWC rule which modifies the create clause in the base query by deleting an existing edge. The LPG created by the mutated query is LPSG-isomorphic to the LPG created by the base query. Rule 12 modifies the base query by wrapping the create clause with a foreach clause, such that the mutated query has a different query execution plan and thus potentially trigger different GDBMS code paths being tested, while the LPG created is isomorphic to the LPG create by the base query.

Modify Return Clause (MRC) Rules. This class of rules does not directly modify writing clauses, yet they modify the return clause in a statement containing writing clauses, with the purpose of introducing more diverse syntax features into the query and thus potentially triggering more bugs. The MRC rules transform a base query into a mutated query that creates identical LPGs with the base query, i.e., satisfies the LPG-isomorphism relation, yet the return different results. For instance rule 20 in Table 2 adds distinct restrictions on return results, which obtaining a mutated query that does not change the graph, yet return less or equal number of records than the base query.

For each seed query, we parse it to identify potential mutation points and apply a mutation rule that fits the point based on its probability. If no rule is identified, a random one is chosen. We support multi-step-mutation when mutation rules are transitive, meaning consecutive applications of two mutation rules that both adhere to Labeled Property Graph Isomorphism will also comply.

4.3 Oracle

Since we focus on detecting bugs triggered by queries containing writing operations, which change the graph database status, the oracle should precisely capture those changes. Moreover, the queries may return records, which should also be correct according to the query semantics. Therefore, our oracle contains two parts, i.e., checking the correctness of the graph database status and checking the correctness of the query results.

Graph-state persistence. To check whether a query correctly modifies the graph database, we rely on the metamorphic relations, i.e., LPG-isomorphism and LPSG-isomorphism, proposed in Section 3. In particular, we propose three classes of mutation rules (in section 4.2) guided by the isomorphism relations and then generate a pair of base query and mutated query, such that the graph databases modified by the pair of queries preserve the corresponding isomorphism relation. Therefore, the problem of checking whether a query correctly modifies the graph database is transformed into the problem of checking whether the two graph databases (which are labeled property graphs) updated by the pair of base and mutated queries preserve the LPG-isomorphism or LPSG-isomorphism relations, we refer this checking as the graph-state persistence checking.

Deciding whether two graphs are isomorphic is a known NP problem in the graph theory and there are various algorithms [14, 37] trying to solve the problem, where VF2 is the most widely adopted due to its stable performance on different types of graphs

Algorithm 1: Deciding Isomorphism Relations of two Labeled Property Graphs (LPG)

```
Input  : two LPGs $G_1$, $G_2$; an intermediate state $s$; the initial state
         $s0$ has $M(s0) = \emptyset$
Output : the mappings between the two graphs or mapping failed
1  Function F($s, n, m$):
2      if Nodes n and m have identical labels, properties, outgoing edges
         and incoming edges then
3          return True
4      end
5      return False
6  Function ComputePairs($s, G_1, G_2$):
7      $P(s)$={}
8      Let $T_1^{out}(s)$ and $T_2^{out}(s)$ be the sets of nodes that are the
         destination of edges starting from $G_1(s)$ and $G_2(s)$
9      Let $T_1^{in}(s)$ and $T_2^{in}(s)$ be the sets of nodes that are the origin
         of edges ending in $G_1(s)$ and $G_2(s)$
10     foreach $n$ in $T_1^{out}(s)$, $m$ in $T_2^{out}(s)$ do
11         Add $(n, m)$ to $P(s)$
12     end
13     foreach $n$ in $T_1^{in}(s)$, $m$ in $T_2^{in}(s)$ do
14         Add $(n, m)$ to $P(s)$
15     end
16     return $P(s)$
17 Function MATCH($s$):
18     if $M(s)$ covers all the nodes of $G_2$ then
19         return $M(s)$
20     end
21     else
22         $P(s)$=ComputePairs($s, G_1, G_2$)
23         foreach $(n, m)$ in $P(s)$ do
24             if F($s, n, m$) then
25                 Compute the state $s'$ obtained by adding $(n, m)$ to
                     $M(s)$
26                 CALL MATCH($s'$)
27             end
28         end
29         Restore data structures
30     end
```

- [16]. Therefore, we adopt VF2 and customize it according to our specific application scenario.

Algorithm 1 shows our customized algorithm for deciding the LPG-isomorphism of two labeled property graphs. The input contains two LPGs G_1, G_2 to be checked, an intermediate state s which is used to record the current mapping status, the initial state s_0 such that $M(s_0)$ is an empty set. Note that $M(s)$ maintains all the node pairs which have been mapped successfully under current state s. The output of our algorithm is either the mapping result or mapping failed. MATCH(s) (lines 17-30) is the main logic of the VF2 algorithm, which returns the graph mappings if all nodes in both graphs are successfully mapped (lines 18-20). Otherwise, it first computes $P(s)$ by invoking function ComputePairs (line 22), which contains all node pair candidates to be added to the $M(s)$. Then each pair in $P(s)$ is checked for feasibility of adding to $M(s)$, which is accomplished by function F(s, n, m) (line 24). Whenever a new pair is added to $M(s)$, a new status s' is created, which calls MATCH(s') recursively for subsequent mappings (lines 24-27). Line 29 is the backtracking process when no pairs in $P(s)$ can be added to $M(s)$. Function ComputePairs (lines 6-16) basically add all node pairs that are one-step reachable to nodes in successfully mapped pairs in $M(s)$ into $P(s)$.

Our algorithm follows the main logic of VF2, with the following customization, which simplifies the complexity of applying the algorithm in our scenario. In the original VF2 algorithm, which applies to general graph matching scenarios, the function that decides whether the node pairs in $P(s)$ can be added to $M(s)$ by mainly checking on the in-degree and the out-degree of the corresponding nodes. In our scenario, we work with labeled property graphs, where nodes and edges are associated with labels and properties, which should be the same according to our definition on LPG-isomorphism. Therefore, we add a semantic checking step to check the labels and properties of the corresponding nodes of the pair.

Note that the original VF2 algorithm has a worst case complexity of $O(n!n)$ [14]. Our scenario is the simplified scenario of the original graph isomorphism checking problem, since the LPGs that we need to check on are created incrementally by our queries, and thus the status s in the algorithm can be associated with each generated query. We implemented the above customized VF2 algorithm that assigns a unique ID property to each node and edge, this approach streamlines our search process by eliminating the need for backtracking unsuccessful mappings, reducing the complexity to $O(n^2)$, which is the best case complexity of VF2 [14]. Our experiments indicate that LPG comparison accounts for only 2.64% of GraspDB's total execution time.

For LPSG-Isomorphism checking, as we can obtain the nodes/edg -es or properties being modified on G' by the mutated query, we can easily create a sub-graph or super-graph $\bar{G}$ base on G, which is the LPG queried by the base query, by applying the modifications to G. Then we invoke the LPG-Isomorphism checking algorithm for the graph pair of (G', $\bar{G}$), which should be LPG-Isomorphic.

Query result correctness. Since the queries with writing clauses may also have return results, we need also check the correctness of the query results as part of our oracle to reduce potential false negatives, i.e., the cases where graph-state is correctly preserved yet the return results are incorrect. This step simply check whether the returned result sets satisfy the relations (=, ⊂, ⊃) introduced by the corresponding mutation rule. Note that different from existing approaches on testing GDBMS [21, 25], where only identical result sets are regarded as correct, our mutation rules may generate query pairs which result in different result sets (e.g., rules 18-20 in Table 2), making our oracle able to detect bugs that are not detectable by oracles of existing approaches.

Table 3: Information on The Tested GDBMS.

GDBMS	Rank	Github Stars	Init Release	LoC
Neo4j	1	12.4k	2007	1,304K
RedisGraph	-	2.0k	2018	1,298K
MemGraph	8	2.1k	2017	268K
AgensGraph	30	1.3k	2016	1,510K

5 Evaluations

We implement our prototype GraspDB with over 11K non-comment lines of Java code. GraspDB uses Neo4j Java Driver 4.1.1 to connect and interact with Neo4j and Memgraph, JRedisGraph 2.5.1 to connect and interact with RedisGraph and AgensGraph Java Driver 1.4.2 to connect and interact with AgensGraph. All evaluations are conducted on a computer with Intel i5-8400 CPU, 16 GB of memory and Windows 11 OS. We aim to address the following research questions in our evaluation.

- **RQ1:** Can GraspDB detect unknown real-world bugs?
- **RQ2:** How does each component contribute to the overall effectiveness of GraspDB?
- **RQ3:** How does GraspDB perform in bug detection effectiveness compared to baselines?

5.1 Evaluation Setup

Testing Subjects. We select four popular real-world graph database engines that support Cypher as our testing subjects. Table 3 shows the meta information of the tested graph databases. Neo4j [10] is the most widely adopted graph data platform in the market according to the DB-Engines Ranking [4]. RedisGraph [11] is a high-performance graph database module that extends Redis. It employs the property graph model and uses the Cypher query language for data manipulation and retrieval. RedisGraph is known for its high performance and real-time data processing capabilities. Memgraph [8] is an in-memory graph database designed for real-time data analytics. It supports openCypher and is compatible with Neo4j. AgensGraph [2] is based on the powerful PostgreSQL RDBMS, and is optimized for handling complex connected graph data and provides plenty of powerful database features essential to the enterprise database environment. We test the Neo4j Community Edition from v5.6.0 to v5.12.0, RedisGraph from v2.10.9 to v2.12.10, MemGraph Community Edition from v2.7.0 to v2.11.0 and AgensGraph Community Edition 2.13.1. All these versions are latest during our testing period from 2023.4 to 2023.10.

Baselines. We compare GraspDB with GDSmith [20] and GraphGenie [21]. GDSmith is the latest differential testing approach on graph databases that is capable of generating syntax and semantic correct Cypher queries. GraphGenie is an metamorphic testing

Table 4: Bugs Detected by GraspDB.

GDBMS	Detected	Confirmed	Fixed	Duplicate	Writing-related	GSO-related
Neo4j	35	33	30	2	18	2
RedisGraph	15	12	6	1	6	4
MemGraph	17	13	7	0	7	4
AgensGraph	10	0	0	0	0	0
Total	77	58	43	3	31	10

Table 5: Types of The Confirmed Bugs

GDBMS	Logic	Error	Crash	Inconsistency
Neo4j	2	29	0	2
RedisGraph	4	2	6	0
MemGraph	5	1	7	0
Total	11	32	13	2

Table 6: Distinct Bugs Detected by Variants of GraspDB

Method	Neo4j	RedisGraph	MemGraph
GraspDB	8	7	10
GraspDB$_{\text{-GSO}}$	7	6	7
GraspDB$_{\text{-AWC}}$	7	6	9
GraspDB$_{\text{-MWC}}$	6	4	6
GraspDB$_{\text{-MRC}}$	6	5	8

approach, which conducts graph pattern transformations guided by injective and surjective relations to generate comparable query pairs. GraphGenie has shown state-of-the-art performance in detecting bugs in GDBMS [21] and outperforms other metamorphic testing approaches such as GDBMeter [22]. Therefore, we choose GDSmith and GraphGenie as baselines for comparison.

5.2 RQ1: Ability on Detecting Unknown Bugs

Table 4 show the bugs that GraspDB detected on the four testing subjects. GraspDB detected 77 bugs during the testing period of around 6 months. Among those bugs, 58 are confirmed and 43 have been fixed. Among the confirmed bugs, 31 is detected by writing clauses in the generated queries, 10 are detected by our graph-state persistent oracle and and 13 via our mutation rules.

Table 5 shows the types of the confirmed bugs. We can observed that the majority of bugs cause errors or system crashes, indicating that the writing-related functionalities are under-tested, and that those bugs may cause serious results, such as system crash. It is thus critical to detect bugs in writing-related functionalities. There are 11 logical bugs, 10 of which are detected by the graph-state-persistent oracle and the other one is detected by return results comparison. There are 2 bugs in the inconsistency category, meaning that the bugs are caused by GDBMS implementation being inconsistent with Cypher document.

5.3 RQ2: Contribution of Different Components

In order to validate the contribution of different components of GraspDB to bug detection, we remove each component of GraspDB, i.e., the graph state oracle, the AWC mutation rules, the MWC mutation rules and the MRC mutation rules, to obtain 4 variants of GraspDB, which are noted GraspDB$_{-GSO}$, GraspDB$_{-AWC}$, GraspDB$_{-MWC}$, GraspDB$_{-MRC}$, correspondingly. And we run GraspDB and its variants on identical environments (Neo4j v5.8.0, RedisGraph v2.12.10 and MemGraph v2.10.0) separately for 12 hours each, more than 360 bug reports are produced for each variant. We randomly selected 10% of the bug reports to analyze due to the heavy manual efforts required for the large amounts of bug reports.

Table 6 reports the experiment results of distinct bugs detected by each variant. We can observe that GraspDB detects the most number of distinct bugs during the experiment period. The variant of removing the graph-state oracle (GraspDB$_{-GSO}$), and variants of removing each class of mutation rules (GraspDB$_{-AWC}$, GraspDB$_{-MWC}$, GraspDB$_{-MRC}$) all miss to detect some of the bugs. For instance, GraspDB$_{-GSO}$ is not able to detect logic bugs caused by writing operations. GraspDB$_{-MRC}$ cannot detect bugs that causes return results to be different. We can conclude from the experiment results that all components of GraspDB contribute individually to the effectiveness of detecting bugs and thus it is the most effective to combine all of them for bug detection.

5.4 RQ3: Comparison with Baselines

Although it is difficult to have a fair and direct comparison between testing techniques, we conducted a best-effort empirical testing comparison between GraspDB and baselines to illustrate their differences. In particular, we run each compared tools in the same environment for the same duration of time, and compare on the distinct bugs detected as well as the false alarm rate. Following the experiment settings of GDSmith [20], we ran GraspDB, GDSmith and GraphGenie on the databases supported by them for 12 hours. All databases were the newest version at the time of the experiment, with Neo4j v5.12.0, RedisGraph v2.12.10, MemGraph v2.11.0 and AgensGraph v2.13.1. GDSmith is capable of generating test cases and graph databases. GraspDB improves GDSmith with writing-related clauses and sub-clauses for test case and database generation. GraphGenie does not support database generation, therefore, we follow the setting of GraphGenie and uses the Movie Graph [26], consisting 171 nodes and 253 relationships as the database for the experiment of GraphGenie.

5.4.1 Detected Unique Bugs. Table 7 shows the comparison results with baselines, where we report the total number of bug reports analyzed, the number of unique bugs in those bug reports and the number of false alarms. During the 12-hour testing, 13632, 3052, and 2502 bugs were reported by GDSmith, GraphGenie (auto-deduplicated), and GraspDB, respectively. Due to the lack of reliable bug deduplicators and the demand for identifying false alarms, we made efforts to manually examine the results of all methods in our evaluation to increase the reliability of the findings. Note that GraphGenie's deduplicator works by recognizing bugs triggered by queries mutated from the same base query and with identical return result counts as duplicate, suffering from false positive when different bugs lead to the same return result counts. The situation becomes more problematic in our case as not all captured bugs have return results. Given large number of reported bugs, it is infeasible

Table 7: Results on Comparison with Baselines

Approach	Neo4j			RedisGraph			MemGraph			AgensGraph		
	Total	Unique	False Alarm	Total	Unique	False Alarm	Total	Unique	False Alarm	Total	Unique	False Alarm
GDsmith	250	0	250	250	2	87	250	0	250	-	-	-
GraphGenie	2	0	2	250	3	85	-	-	-	39	0	39
GraspDB	24	3	5	75	6	5	104	6	13	250	10	9

to manually analyze each report. Hence, we randomly sampled 250 reports from each target GDBMS for manual analysis.

Since GDSmith is a differential testing approach and it compares the results of the three tested GDBMSs to report bugs, therefore, the bugs reported in Neo4j, RedisGraph and MemGraph are the same. The results show that there are 2 unique bugs detected from those 250 bug reports. The two bugs are both in RedisGraph, which does not follow the Cypher documentation in their implementation. Those bugs are known limitations [7] and the developers won't fix them for a while for the sake of robustness. As GDSmith is a differential testing approach, which report bugs by comparing all the databases under test, therefore, the reported numbers of the three supported databases (Neo4j, RedisGraph and MemGraph) are the same. GDSmith does not support AgensGraph and thus we do not report the results on it.

GraphGenie produces 2 and 39 bug reports on Neo4j and AgensGraph, respectively, and after our manual analysis, all of the reported bugs on those two databases are false alarms. GraphGenie produces 3052 bug reports on RedisGraph. We randomly sampled 250 bug reports for manual analysis and 3 unique bugs are identified. GraphGenie does not support MemGraph and thus we do not report the results on MemGraph.

GraspDB produced 24, 75 and 104 bug reports and 3, 6 and 6 are distinct bugs on Neo4j, RedisGraph and MemGraph, respectively. GraspDB produced 2502 bug reports on AgensGraph, we randomly selected 250 for manual analysis and identified 10 unique bugs.

To summarize, GraspDB found 15 unique bugs in 12 hours, 8 of which are related to writing operations, while GDSmith and GraphGenie can only find 2 and 3 bugs, respectively, which are all not writing operation-related bugs. On average, 18 bug report is analysed to identify 1 unique bug detected by GraspDB, while this number is 97 and 125 for GraphGenie and GDSmith, respectively. GraspDB can detect bugs more accurately with low duplication rate, and thus requires far less manual efforts for bug report analysis. Overall, GraspDB generates fewer bug reports. One reason is the overhead incurred by the increasingly large graph and the large number of time-consuming writing operations. Less test cases can be examined during the same time periods, hence less bug reports. However, changing graph sizes and writing operations are essential for discovery of writing related bugs. Another reason is due to the low duplicate bug report rate of GraspDB compared to baselines.

5.4.2 False Alarms. GDsmith is a differential testing approach for Cypher-based GDBMSs and found 28 previously unknown bugs. However, we find that it has a high false alarm rate during our preliminary experiments on it. The reason is mainly due to the different implementation choices on the same Cypher feature by different GDBMSs. This is a common limitation for differential testing approaches and similar findings is also reported by GraphGenie [21]. This limitation leads to heavy manual efforts in analyzing the large amount bug reports to rule out false positives. As shown in Table 7, 87 false alarms have been identified from the 250 bug reports of GDsmith, making more than 1/3 of the reported bugs false positives. All of the false alarms are caused by different implementation choices by different GDBMSs, Among which 70 are due to undefined behavior in Cypher documentation, and 17 are due to different numerical representation formats. One such false alarm example is shown in Figure 5(a), where Neo4j and MemGraph have different implementation choices for the return clause which returns aggregate function as results.

```
WITH 1 AS a WHERE false WITH 1 AS n RETURN max(1),n;
--Result on Neo4j: empty
--Result on MemGraph: null,null
```

(a) Example false alarm by GDSmith due to different semantic implementations on GDBMSs

```
OPTIONAL MATCH (c)-[:PP]-() WHERE false RETURN count(c)
--Return Result: 0
OPTIONAL MATCH (c)-[:PP]-() WHERE false RETURN count(*)
--Return Result: 1
```

(b) Example false alarm by GraphGenie due to nonequivalent semantics of Count(c) and Count(*)

```
CREATE (),();
MATCH () UNWIND [1,1] as a WITH DISTINCT * RETURN a;
--Return Result: 1
CREATE (),();
MATCH path=() UNWIND [1,1] as a WITH DISTINCT * RETURN a;
--Return Result: 1,1
```

(c) False alarm of GraspDB due to adding a variable

Figure 5: False alarm examples

GraphGenie is a metamorphic approach, which do not have false alarms introduced by different implementation choices. The false alarms produced by GraphGenie is mainly due to inaccurate transformation rules. Figure 5(c) shows a false alarm produced by GraphGenie. The semantics of `count(c)` and `count(*)` are different, as `null` is not counted in the former case and is counted in the latter case. GraphGenie used them for identical semantic transformation, which caused this false alarm. We also identify another similar incorrectly used clause pairs, i.e., `OPTIONAL MATCH` and `MATCH`, by GraphGenie, which introduces false alarms. We've contacted the authors of GraphGenie and received their confirmation on the reasons of the false alarms.

There are also some false alarms introduced in our approach, with similar causes as GraphGenie. Figure 5(d) shows an example false alarm GraspDB produced. In our mutation rule, a variable p is introduced with the purpose of perceiving the semantics of the query. However, in this case, the semantics of the query is changed. The root cause is that the new variable p is introduced before `WITH` clause, causing more records to be passed backwards when filtering out the same records by clause `WITH DISTINCT *`, leading to wrong return results. This type of false alarms is similar with that of GraphGenie and can be avoided by carefully implement the mutaiton rules to filter out those cases. GraspDB has a far lower false alarm rate compared with GDSmith and GraphGenie, saving plenty of time in manual checking the bug reports.

5.5 Threats to Validity

Threats to internal validity mainly lie in the implementation of GraspDB. The random query generation of GraspDB produces syntactically valid statements, which, however, can result in semantic errors when being executed. For instance, integer adding can result in integer out of bounds. We automatically filtered those semantic errors based on error messages, which may potentially filter bugs in GDBMSs which are not due to semantic errors. And due to the large number of report generated, we have to sample to analyze and identify the bugs, which may miss bugs. Another threat lies in the manual validation process of logic bugs found in GDBMs. Due to the large number of report generated, we have to sample to manually deduplicate and identify the bugs, which may miss bugs. The manual analysis process may also introduce errors. To make the validation as accurate as possible, three authors analyze all reported discrepancies, and reach consensus for all discrepancies. The large number of duplicated bugs poses a significant challenge for manual analysis. Bug automatic deduplication is an important and challenging research topic with many ongoing studies [39]. GraphGenie's deduplicator works by recognizing bugs triggered by queries mutated from the same base query and with identical return result counts as duplicate, suffering from false positive when different bugs lead to the same return result counts. The situation becomes more problematic in our case as not all captured bugs have return results. As a compromisation, we manually deduplicate the bugs by analyzing both the bug symptoms and root causes. We leave automated bug deduplication as a future work.

Threats to external validity mainly lie in the subjects chosen in our evaluations, which do not cover all graph databases supporting Cypher. To mitigate the threat, we pick four well-known, open-source graph database engines with active development community for fast feedback. The selected GDBMSs are among the most popular graph databases and have different designs on system architecture or data storage, making them representative. GraspDB is applicable to test any graph databases supporting the Cypher language. Actually, GraspDB can also be applied to graph database engines supporting other languages such as Gremlin, with the assistant of query translation plugins such as Cypher for Gremlin [3].

6 Selected Bugs

Since the purpose of our approach is primarily to test the writing-related functionalities of graph databases, we analyze the bugs related to writing clauses. For the clarity and simplicity of illustration, we show the minimized test cases as well as minimized graph data necessary to demonstrate the underlying core problem.

```
Create p=()<-[r:T]-() Create ({k:COUNT{RETURN [r]}});
--Expected behavior: 1 path and 1 node created
--Actual behavior: ExecutionFailed Error
```

(a) Consecutive node creation with dependencies failed in Neo4j

```
CREATE (),();
MERGE (n0) MERGE (n1) CREATE (c0);
--Expected behavior: 4 nodes created
--Actual behavior: Memory limit exceeded and Connection closed
```

(b) Nodes are created endlessly in MemGraph

```
CREATE (n0)<-[r0:T]-() DETACH DELETE n0 DETACH DELETE r0;
--Expected behavior: n0 and r0 are deleted
--Actual behavior: Connection closed
```

(c) Crash caused by repeated deleting a relationship in MemGraph

Figure 6: Case study of bugs detected by GraspDB

Consecutive node creation with dependencies failed in Neo4j. Figure 6(a) shows that an error occurs in a query which creates nodes that have dependencies[2]. The root cause is due to an optimization in Neo4j, which merges two consecutive CREATE clauses to be able to create them together. However, this optimization shouldn't apply in the case that the two create clauses have dependencies. Otherwise, an error is triggered due to referencing a variable that hasn't been created yet. We have reported the bug to Neo4j developers and it has been fixed.

Crash caused by creating nodes endlessly in MemGraph. Figure 6(b) shows a query that creates nodes endlessly and eventually leads to a crash and the database connection close[3]. The first line creates 2 nodes in the graph. Then the statements `MERGE (n0)` and `MERGE (n1)` each matches the two nodes created before them, producing 4 rows of records, which are passed to the last create clause, creating 1 node for each matched record and finally result in 4 created nodes. However, due to incorrect semantic implementations by MemGraph, the node creating will not end until exceeding memory limits, and cause connection being closed. This bug seriously affects the usability, and potentially security aspects of the graph database. The developers have confirmed this bug and fixed it for this scenario. They respond that this issue is still open for other scenarios, which they are not able to fix for compatability concerns.

Crash caused by repeated deleting a relationship in MemGraph. Figure 6(c) shows a query that leads to a crash due to deleting a relationship which has been deleted before[4]. According to the OpenCypher9 documentation, `DETACH DELETE` is used to delete nodes (including relationships connected to it) or relationships. Therefore, both n0 and r0 are deleted in the first DETACH DELETE clause. When the second DETACH DELETE clause executes, MemGraph tries to detach relationship r0 from its nodes. Since node n0 is deleted by the previous DETACH DELETE clause, the second delete will crash the database. This bug has already been confirmed and fixed.

[2]https://github.com/memgraph/memgraph/issues/1333
[3]https://github.com/neo4j/neo4j/issues/13305
[4]https://github.com/memgraph/memgraph/issues/1329

7 Related Work

Testing of GDBMS. GDBMSs have gained extensive adoption, leading to a growing focus on their quality and correctness. Two types of approaches have been proposed for testing GDBMS, i.e., differential testing approaches and metamorphic testing approaches.

Grand [41] realized differential testing for GDBMSs that support the Gremlin language. For test case generation, Grand uses a model-based approach to generate valid Gremlin queries. GDsmith [20] is another differential testing approach to test GDBMSs which support the Cypher query language, another popular query language for GDBMS. For test case generation, GDsmith uses skeleton generation and completion to generate semantically valid Cypher queries. The major drawback of differential testing is that bugs can not be detected when different DBMSs suffer from the same ones. Furthermore, differential testing approaches suffer from high false alarm rates due to different implementation choices of GDBMSs, as has been discussed in shown in Section 5.4.

GDBMeter [22] is the first metamorphic testing approach which applied the TLP [30] from relational DBMS to graph DBMS and found 40 unique, previously unknown bugs. GraphGenie [21] proposes injective and surjective Graph Query Transformation (GQT) to detect logic bugs. It leverages graph properties to generate follow-up queries by mutating graph query patterns and detected 25 unknown bugs. GRev [25] adapts Equivalent Query Rewriting (EQR) to GDBMS queries by Random Walking on Abstract Syntax Graph (ASG), an abstraction they proposed to represent query paths. Gamera [42] develops three classes of graph-aware metamorphic relations, i.e., elementary MRs, compound MRs and dynamic MRs, which directly applies to labeled property graphs, for testing GDBMS. Different from GraspDB, non of existing metamorphic testing approaches focus on detecting bugs caused by writing operations, and the oracles in their approach are not able to detect labeled property graph changes, thus unable to capture those bugs.

Metamorphic Testing of RDBMS. Ternary Logic Partitioning (TLP) [30], a metamorphic testing approach first proposed for testing RDBMSs, has also been applied to test GDBMS [22]. This tool has been used to find 175 bugs in widely deployed RDBMSs. (PQS) is a general and highly-effective approach to finding bugs in DBMS. The core idea of PQS [31] is to automatically synthesize queries which guarantees to fetch a specific, randomly selected row, called the pivot row. If the DBMS fails to fetch the pivot row, it likely causes a bug in the RDBMS. Non-optimizing Reference Engine Construction (NoREC) [29] is another widely-known metamorphic testing approach to test RDBMS. It compares the execution results of a given optimized query with the non-optimized version that they rewritten based on the original query, to detect optimization bugs in DBMS. The metamorphic testing approaches on RDBMSs share similar purposes and scenarios as GDBMSs, and can potentially be applicable to GDBMSs.

8 Conclusion

In this paper, we present GraspDB, the first metamorphic testing approach specifically designed to identify bugs related to writing operations in graph database systems. In particular, we define the concepts of Labeled Property Graph Isomorphism (LPG-Isomorphism) and Labeled Property Subgraph Isomorphism (LPSG-Isomorphism) relations, serving as the basis of our Graph State-Persistence oracle. To engage more code logic on writing operations, we propose three classes of mutation rules, i.e., add writing clauses, modify writing clauses and modify return clauses, which are guided by the LPG-Isomorphism and LPSG-Isomorphism metamorphic relations. We apply the mutation rules on base queries to obtain base query, mutated query pairs, the execution correctness of which can be verified with the metamorphic relations. We conduct experiments on four commercial GDBMSs. Our approach detected 77 previous unknown bugs. 58 of them have been confirmed by the developers, and 43 have been fixed. Among them, there are 11 logic bugs, 32 errors, 13 crashes and 2 inconsistencies with Cypher documentation. Additionally, 31 bugs cannot be triggered without database writing operations. We have implemented our method as a tool called GraspDB and made it public available to inspire future research.

9 Data Availability Statement

We have released the source code and data of our work [24] to inspire future research.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant Nos. 62472429, 61972403, 61732014, and by Ant Group through CCF-Ant Research Fund No. CCF-AFSG RF20240103. We thank the anonymous reviewers for their constructive suggestions, which help improve the quality of this paper.

References

[1] 2007. *Structured Query Language.* Springer Berlin Heidelberg, Berlin, Heidelberg, 111–212. https://doi.org/10.1007/978-3-540-48399-1_4
[2] 2024.03. *AgensGraph Official Website.* https://bitnine.net/agensgraph/.
[3] 2024.03. *Cypherforgremlin Plugin.* https://github.com/opencypher/cypher-for-gremlin/tree/master/tinkerpop/cypher-gremlin-server-client.
[4] 2024.03. *The DBRanking Website.* https://db-engines.com/en/ranking/graph+dbms/.
[5] 2024.03. *Graph Database market share.* https://www.marketsandmarkets.com/Market-Reports/graph-database-market-126230231.html..
[6] 2024.03. *Graph isomorphism problem for labeled graphs, computer science.* https://cs.stackexchange.com/questions/28714/graph-isomorphism-problem-for-labeled-graphs.
[7] 2024.03. *Limitations of RedisGraph.* https://github.com/RedisGraph/RedisGraph/blob/master/docs/docs/known_limitations.md.
[8] 2024.03. *MemGraph Official Website.* https://memgraph.com/.
[9] 2024.03. *Neo4j Official Documents: Clauses in the Cypher Query Language.* https://neo4j.com/docs/cypher-manual/current/clauses/.
[10] 2024.03. *Neo4j Official Website.* https://neo4j.com/product/neo4j-graph-database/.
[11] 2024.03. *RedisGraph Official Website.* https://docs.redis.com/latest/stack/deprecated-features/graph/.
[12] Asim Ansari, Skander Essegaier, and Rajeev Kohli. 2000. Internet recommendation systems. https://doi.org/10.1509/jmkr.37.3.363.18779
[13] 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. https://doi.org/10.1145/3448016.3457545
[14] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 10 (2004), 1367–1372. https://doi.org/10.1109/TPAMI.2004.75
[15] 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. https://doi.org/10.1145/3308558.3313488
[16] Pasquale Foggia, Carlo Sansone, Mario Vento, et al. 2001. A performance comparison of five algorithms for graph isomorphism. In *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition.* Citeseer, 188–199. https://src.acm.org/binaries/content/assets/src/2009/sara-voss.pdf

[17] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Martin Schuster, Petra Selmer, et al. 2018. Formal semantics of the language cypher. *arXiv preprint arXiv:1802.09984* (2018). https://dblp.org/rec/journals/corr/abs-1802-09984.html

[18] 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. https://doi.org/10.1145/3183713.3190657

[19] Shu-Ming Hsieh, Chiun-Chieh Hsu, and Li-Fu Hsu. 2006. Efficient method to perform isomorphism testing of labeled graphs. In *Computational Science and Its Applications-ICCSA 2006: International Conference, Glasgow, UK, May 8-11, 2006, Proceedings, Part V 6*. Springer, 422–431. https://doi.org/10.1007/11751649_46

[20] 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 the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 163–174. https://doi.org/10.1145/3597926.3598046

[21] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland Hock Chuan Yap, Zhenkai Liang, and Manuel Rigger. 2023. 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)*. IEEE Computer Society, 531–542. https://doi.org/10.1145/3597503.3623307

[22] Matteo Kamm. 2022. *Testing Graph Databases using Predicate Partitioning*. Master's thesis. ETH Zurich. https://doi.org/10.1145/3597926.3598044

[23] MAO Linfan. 2016. *Labeled graph—A mathematical element*. Infinite Study.

[24] Shuang Liu, Junhao Lan, Xiaoning Du, Jiyuan Li, Wei Lu, Jiajun Jiang, and Xiaoyong Du. 2024. Testing Graph Database Systems with Graph-State Persistence Oracle. Zenodo. https://doi.org/10.5281/zenodo.12670528

[25] Qiuyang Mang, Aoyang Fang, Boxi Yu, Hanfei Chen, and Pinjia He. 2024. Testing Graph Database Systems via Equivalent Query Rewriting. (2024). https://doi.org/10.1145/3597503.3639200

[26] Neo4j. 2024.03. *Recommendation Graph*. https://github.com/neo4j-graph-examples/recommendations/.

[27] Neo4j official. 2023.12. *Cypher Query Language*. https://neo4j.com/developer/cypher/.

[28] Paul Resnick and Hal R Varian. 1997. Recommender systems. *Commun. ACM* 40, 3 (1997), 56–58. https://doi.org/10.1145/245108.245121

[29] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1140–1152. https://doi.org/10.1145/3368089.3409710

[30] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. https://doi.org/10.1145/3428279

[31] 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)*. 667–682. https://doi.org/10.5555/3488766.3488804

[32] Carlos R Rivero and Hasan M Jamil. 2017. Efficient and scalable labeled subgraph matching using SGMatch. *Knowledge and Information Systems* 51, 1 (2017), 61–87. https://doi.org/10.1007/s10115-016-0968-2

[33] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc.". https://doi.org/10.5555/2846367

[34] 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. https://doi.org/10.1145/2815072.2815073

[35] Marko A. Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language (Invited Talk). In *Proceedings of the 15th Symposium on Database Programming Languages* (Pittsburgh, PA, USA) *(DBPL 2015)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2815072.2815073

[36] Christos Tjortjis. 2023. *Graph Databases: Applications on Social Media Analytics and Smart Cities*. CRC Press. https://doi.org/10.1201/9781003183532

[37] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (jan 1976), 31–42. https://doi.org/10.1145/321921.321925

[38] 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. https://doi.org/10.1145/1772690.1772790

[39] Chen Yang, Junjie Chen, Xingyu Fan, Jiajun Jiang, and Jun Sun. 2023. Silent Compiler Bug De-duplication via Three-Dimensional Analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 677–689. https://doi.org/10.1145/3597926.3598087

[40] 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. https://doi.org/10.1109/SP46214.2022.9833669

[41] 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. https://doi.org/10.1145/3533767.3534409

[42] Zeyang Zhuang, Penghui Li, Pingchuan Ma, Wei Meng, and Shuai Wang. 2023. Testing Graph Database Systems via Graph-Aware Metamorphic Relations. *Proceedings of the VLDB Endowment* 17, 4 (2023), 836–848. https://doi.org/10.14778/3636218.3636236

Received 2024-04-12; accepted 2024-07-03