Testing Graph Database Systems via Equivalent Query Rewriting

Qiuyang Mang*
qiuyangmang@link.cuhk.edu.cn
School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China

Aoyang Fang*
aoyangfang@link.cuhk.edu.cn
School of Science and Engineering, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China

Boxi Yu
boxiyu@link.cuhk.edu.cn
School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China
Shenzhen Research Institute of Big Data, China

Hanfei Chen
hanfeichen@link.cuhk.edu.cn
School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China

Pinjia He†
hepinjia@cuhk.edu.cn
School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China
Shenzhen Research Institute of Big Data, China

ABSTRACT

Graph Database Management Systems (GDBMS), which utilize graph models for data storage and execute queries via graph traversals, have seen ubiquitous usage in real-world scenarios such as recommendation systems, knowledge graphs, and social networks. Much like Relational Database Management Systems (RDBMS), GDBMS are not immune to bugs. These bugs typically manifest as logic errors that yield incorrect results (*e.g.*, omitting a node that should be included), performance bugs (*e.g.*, long execution time caused by redundant graph scanning), and exception issues (*e.g.*, unexpected or missing exceptions).

This paper adapts Equivalent Query Rewriting (EQR) to GDBMS testing. EQR rewrites a GDBMS query into equivalent ones that trigger distinct query plans, and checks whether they exhibit discrepancies in system behaviors. To facilitate the realization of EQR, we propose a general concept called *Abstract Syntax Graph (ASG)*. Its core idea is to embed the semantics of a base query into the paths of a graph, which can be utilized to generate new queries with customized properties (*e.g.*, equivalence). Given a base query, an ASG is constructed and then an equivalent query can be generated by finding paths collectively carrying the complete semantics of the base query. To this end, we further design Random Walk Covering (RWC), a simple yet effective path covering algorithm. As a practical implementation of these ideas, we develop a tool *GRev*, which has successfully detected 22 previously unknown bugs across 5 popular GDBMS, with 15 of them being confirmed. In particular, 14 of the detected bugs are related to improper implementation of graph data retrieval in GDBMS, which is challenging to identify for existing techniques.

KEYWORDS

Graph databases, Metamorphic testing, Query rewriting

ACM Reference Format:
Qiuyang Mang, Aoyang Fang, Boxi Yu, Hanfei Chen, and Pinjia He. 2024. Testing Graph Database Systems via Equivalent Query Rewriting. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639200

1 INTRODUCTION

Graph Database Management Systems (GDBMS) are designed for storing and executing queries on graph-structured data. Compared with *Relational Database Management Systems (RDBMS)*, which store data in tables and columns, GDBMS maintain data in a graph model consisting of nodes and edges, representing entities and their relationships. GDBMS provide powerful graph traversal and pattern-matching capabilities, allowing for flexible and efficient query of interconnected data. These features make GDBMS particularly useful and efficient in data-driven applications, such as recommendation systems, knowledge graphs, and social networks. For example, Facebook utilizes its own GDBMS named TAO [3] to store and manage billions of users and their relationships in social networks. According to DB-Engines Ranking [6], there are 51 widely-used GDBMS, such as Neo4j [26], TinkerGraph [41], MemGraph [23], and NebulaGraph [24].

Like any other software, GDBMS are not immune to bugs, including logic bugs leading to incorrect query results, performance bugs causing sub-optimal query execution time or even a long time hang-up, and exception issues such as unexpected and missing exceptions. These bugs can manifest in various functionalities of GDBMS, encompassing (1) *graph-related* bugs that arise from improper implementation of graph data retrieval in GDBMS (*e.g.*, graph pattern matching) and (2) common bugs related to handling

*Both authors contributed equally to this research.
†Pinjia He is the corresponding author.

ICSE 2024, April 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639200

Listing 1: A graph-related bug in RedisGraph detected by our approach, where two equivalent queries result in inconsistent counting results.

```
(1) MATCH (A)-[]->(B :L1), (C)<-[]-(A)
MATCH (C)<-[]-(D :L2)-[]->(D :L2) RETURN COUNT(*) --{7}

(2) MATCH (C)<-[]-(A)-[]->(B :L1)
MATCH (C)<-[]-(D :L2)-[]->(D :L2) RETURN COUNT(*) --{8}
```

functionalities similar to those in RDBMS (*e.g.*, predicates). Listing 1 demonstrates a graph-related logic bug in RedisGraph [31] that was detected by our approach.[1] In the first query, the relationships between nodes (A) and (B) are retrieved first, and then the relationship between (A) and (C). In the second query, both relationships are retrieved simultaneously. RedisGraph returns different results for these two queries even though they query the same graph data. This inconsistency indicates the presence of a logic bug during the graph data retrieval in RedisGraph.

Detecting bugs in GDBMS is very challenging because of the following reasons. First, unlike RDBMS which uniformly utilize SQL to access data, different GDBMS adopt different query languages, such as Neo4j's Cypher [28], TinkerGraph's Gremlin [35], and NebulaGraph's nGQL [27]. Thus, existing differential testing tools, like GDsmith [13] and Grand [44], can only test the shared functionalities of GDBMS that adopt the same query language, such as Neo4j's feature of label expressions [25]. Moreover, the adoption of different query languages can lead to extensive engineering effort when developers try to generalize existing tools to more query languages. We think this is the main reason why existing research focuses on at most two query languages: Cypher and Gremlin.

Second, it is non-trivial to adapt RDBMS testing approaches [2, 32, 33, 36] to GDBMS because it incurs re-implementation under different database models and query languages. Moreover, this kind of adaptation can only test the shared functionalities between RDBMS and GDBMS (*e.g.*, expressions and predicates), leaving graph-related bugs underexplored. For example, GDBMeter [15], the only existing metamorphic testing approach for GDBMS, utilizes an RDBMS testing approach *Ternery Logic Partitioning (TLP)* [32], which partitions a query Q into sub-queries $Q_P, Q_{\neg P}, Q_{\text{null}}$ based on three possible outcomes of the predicate P: true, false, and null. Thus, although GDBMeter found common bugs related to *predicates* handling, none of their reported bugs are graph-related.

Third, performance bugs in GDBMS have not been carefully considered by existing approaches. Specifically, differential testing approaches, such as GDsmith [13] and Grand [44], fall short of detecting performance bugs because the diverse underlying architectures of different GDBMS can easily cause performance differences. The metamorphic testing approach, GDBMeter [15], is based on the relationship between the original query and three sub-queries, which do not exhibit clear performance relationships. Thus, it is difficult for existing approaches to find performance bugs.

To tackle these challenges, this paper adapts *Equivalent Query Rewriting (EQR)* to GDBMS testing. For example, the first query in Listing 1 was re-written to generate the second query as the equivalent query. If the returned results are different or a large difference was observed in the query time, a potential bug is found. To facilitate the realization of EQR in GDBMS, we propose a novel, widely applicable concept called *Abstract Syntax Graph (ASG)*. The key insight is to embed the complete semantics of a base query into the paths of a graph (*i.e.*, ASG). An ASG can be further utilized to generate queries with customized properties (*e.g.*, equivalence) by covering different sets of paths in the graph automatically. Intuitively, ASG provides an abstraction for a query that can be mapped to various concrete graph models adopted by different GDBMS and their query languages. Thus, to generate an equivalent query that carries the complete semantics encoded in ASG, we need to find a set of non-overlapping paths that collectively cover all the relationships (*i.e.*, edges) and constraints (*i.e.*, labels and properties). This path-finding process is further handled by our proposed *Random Walk Covering (RWC)*, a simple yet effective algorithm. Each set of the non-overlapping paths returned by RWC will be translated into an equivalent query. RWC can generate a large number of equivalent queries that trigger diverse query plans, and thus effectively stress-test GDBMS. Unlike traditional query rewriting techniques [21] that rely on multiple hardcoded rules, our methods do not require domain experts on query languages nor extra engineering resources for identifying and verifying various mutation rules. In addition, ASG and RWC can be easily adapted to different query languages, which only incurs lightweight parser development. Differently, traditional query rewriting needs to re-identify mutation rules to conform to different query language grammar.

We have implemented these ideas as a tool called *GRev*, and employed it to test five popular GDBMS, namely Neo4j [26], RedisGraph [31], MemGraph [23], TinkerGraph [41], and NebulaGraph [24], which collectively adopt three different query languages: Cypher [28], Gremlin [35], and nGQL [27]. As of the time of writing, GRev has discovered 22 previously unknown bugs in these GDBMS, including 14 graph-related bugs, of which 15 have been confirmed. Among the 22 bugs, 12 are logic bugs, 3 are performance bugs, 4 are missing exceptions, and 3 are unexpected exceptions.

In summary, this paper makes the following contributions.

- We adapt *Equivalent Query Rewriting (EQR)* to GDBMS testing.
- We propose *Abstract Syntax Graph (ASG)*, a general concept that encodes the complete semantics of a query in a graph.
- We design a simple yet effective algorithm *Random Walk Covering (RWC)* to generate equivalent queries from ASG.
- We implement these ideas as a tool called *GRev* and use it to test 5 widely-used GDBMS, and successfully detect 22 previously unknown bugs with 15 confirmed and 14 graph-related.

2 BACKGROUND

2.1 Labeled Property Graph in GDBMS

Most GDBMS adopt a *labeled property graph (LPG)* to store and query data. As shown in Fig. 1, an LPG contains nodes (*e.g.*, n1, n2 and n3), relationships (*e.g.*, r1 and r2), corresponding labels (*e.g.*, n1 : person and r1 : read), and properties (*e.g.*, name : Alice) for each node and relationship. The flexible graph structure can efficiently capture interconnected information regarding entities, relationships, and attributes, whereas RDBMS stores this information using

[1] https://github.com/RedisGraph/RedisGraph/issues/3093

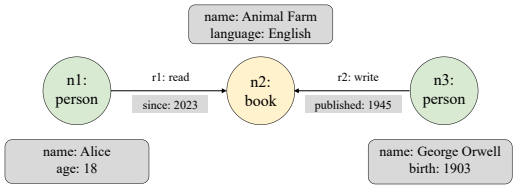


Figure 1: A labeled property graph (LPG) example, which contains three nodes (n1, n2 and n3), two relationships (r1 and r2) and their corresponding labels (*e.g.*, n1 : person) and properties (*e.g.*, age : 18)

several tables. Owing to its flexibility and effectiveness in representing interconnected data, the LPG model has become a crucial component of GDBMS.

2.2 Query Languages in GDBMS

Unlike RDBMS, which utilizes SQL for creating, modifying, and retrieving data, graph databases do not have a standardized query language. Instead, multiple query languages have been developed for different GDBMS to cater to various application requirements. For instance, Cypher [28] is a popular SQL-like query language that employs SQL-like clauses such as `MATCH`, `WHERE`, and `RETURN` to retrieve data. On the other hand, Gremlin [35] is a widely-used functional query language that expresses queries in a data-flow manner. Among the top 10 GDBMS according to the DB-Engine Ranking [6], 6 of them support Gremlin or Cypher as their query language. Therefore, we realize EQR for mainly validating Cypher-based and Gremlin-based GDBMS. In addition, we tested NebulaGraph [24] that utilizes their own language nGQL [27] to further verify the effectiveness and generalizability of our approach.

2.3 Query Rewriting

Query rewriting [30] is a crucial step in RDBMS query optimization, which primarily aims at transforming an original query into an equivalent but more efficient form to speed up execution. This transformation is facilitated by a set of hardcoded rewrite rules, which are selected and applied using a production rule engine, significantly improving the efficiency of query execution. AMOEBA [21] utilizes query rewriting to identify performance bugs in RDBMS, but it still necessitates the manual definition of rewriting rules by domain experts.

While inspired by Query Rewriting, our EQR pursues a different goal, which is to generate numerous equivalent queries automatically with diverse query plans without any hardcoded rules. In addition, existing query rewriting rules explored in the database field cannot be used in GDBMS testing because of the following reasons. First, most query rewriting studies focus on RDBMS and there is a lack of similar techniques in GDBMS. Second, even if researchers start to explore rewriting rules in graph databases for query optimization, which is a challenging research direction, the equivalent queries generated by them are unlikely to trigger diverse query plans, because these potential rules would be quickly integrated into the optimizers of graph databases and the equivalent queries generated by them will be compiled into the same query

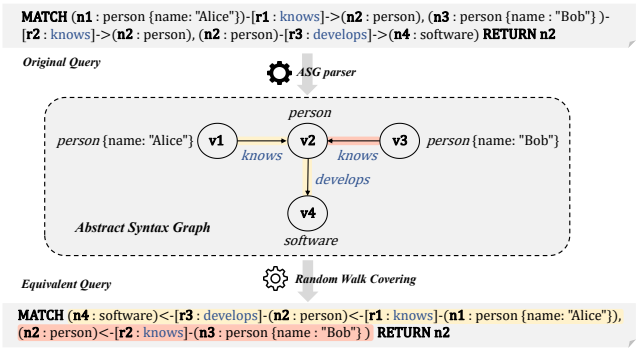


Figure 2: An illustrative overview of utilizing our Equivalent Query Rewriting (EQR) approach to rewrite a concrete query (*i.e.*, finding software developers known by Alice and Bob) while ensuring the equivalence.

plan. As a result, these methods may not work when newer versions of DBMS adopt the rewrite rules. Differently, our proposed EQR is a general methodology for validating GDBMS (different goal) and is able to generate large numbers of equivalent queries (explained in Section 3.3), which cannot be easily optimized to the same query plan, thus solving the previous problems.

3 APPROACH

To tackle the challenge of realizing ERQ in GDBMS, we introduce a general concept called *Abstract Syntax Graph (ASG)* (Section 3.1). The core insight is to encode the complete semantics of the query into a set of paths that form a graph (*i.e.*, ASG). Then an equivalent query can be generated by finding another set of paths that cover all the relationships (*i.e.*, edges) and constraints (*i.e.*, labels and properties) in the ASG. To this end, we also design a simple but effective path-finding algorithm called *Random Walk Covering (RWC)* (Section 3.3). RWC can traverse the ASG in different ways and return different sets of paths, which will be further translated into different equivalent queries.

The EQR process, as illustrated in Fig. 2, begins with an ASG parser transforming the concrete query into its corresponding ASG representation (Section 3.2). After that, the ASG is fed into the RWC algorithm (Section 3.3), which generates multiple equivalent queries. In the following, introduce ASG, ASG parsing, and RWC in detail.

3.1 Abstract Syntax Graph (ASG)

Various graph database query languages can offer distinct functionalities, cater to different usage scenarios, and employ unique syntax. This diversity presents a significant challenge when attempting to propose a method that supports rewriting equivalent queries for different query languages universally. Despite this challenge, graph databases generally share the same fundamental goal of searching for graph components that have certain relationships between nodes.

Based on this insight, we propose ASG, a general concept that can represent the semantics of a query, *i.e.*, relationships between

nodes to be retrieved by the queries and constraints around the edges or nodes, in a language-independent manner. ASG is not specifically tailored to any particular query language, which makes it a flexible and generic representation for various queries. Specifically, to extend our tool to a new graph query language, developers only need to provide a translator that translates between the query language and ASG, rather than writing full-stack code from scratch.

Given a query Q, an ASG is a directed graph structure that encapsulates the relationships between nodes to be retrieved by Q. In particular, these relationships are defined as patterns in Cypher [28] and traversals in Gremlin [35]. In a rigorous formalization, the ASG of Q is denoted as $G(Q) = (V, E)$, where

- V represents a set of vertices. Each vertex corresponds to a category of nodes to be retrieved by Q, aligned with their specific *constraints* defined in Q (*e.g.*, they must carry certain labels, or their property values have to satisfy a certain inequality).
- E is a set of edges, where each edge links two vertices within V, describing the relationship that should be satisfied between these two categories of nodes.

For instance, consider the ASG in Fig. 2, which illustrates the relationships between the nodes to be retrieved. This ASG comprises four vertices, namely `v1`, `v2`, `v3`, and `v4`. These vertices correspond to four categories of nodes—`n1`, `n2`, `n3`, and `n4`—to be retrieved, each paired with specific constraints. As an example, `v1` aligns with the category of nodes labeled as `person`, with the `name` property being "`Alice`". Additionally, there are three edges in the ASG: `v1`→`v2`, `v2`←`v3`, and `v2`→`v4`. These edges illustrate the relationships between the categories of nodes. For example, the edge `v1`→`v2` indicates that `n1` should know `n2` for each tuple (`n1`, `n2`, `n3`, `n4`) among the retrieved data.

3.2 ASG Parsing

The goal of ASG parsing is to convert the data relationships of queries, as expressed in the query languages, into a graph representation within the ASG. This step incurs lightweight engineering effort so our method can be easily adapted to other query languages. In particular, we use around 260 lines of Python code for Cypher [28] ASG parsers. These parsers translate the diverse syntax and semantics of different query languages into a unified ASG representation. Specifically, when a GDBMS query aims to search multiple graph components simultaneously (*e.g.*, simultaneously finding Alice's friends and Bob's friends in the social networks), this translation also allows for the potential division of a complex query into multiple ASGs, where each ASG represents a distinct section of the query.

In our implementation, for Cypher queries, each pattern within the query is extracted and converted into an individual ASG; and for Gremlin queries, the start step, end step, and each `as()` step are converted into ASG vertices, while the inner traversal steps are transformed into ASG edges.

3.3 Random Walk Covering

Given the necessity of expressing queries in linear text, relationships that need to be retrieved are commonly represented by multiple path-like relationships within a GDBMS query. For instance, a Cypher query's graph pattern consists of a series of path patterns,

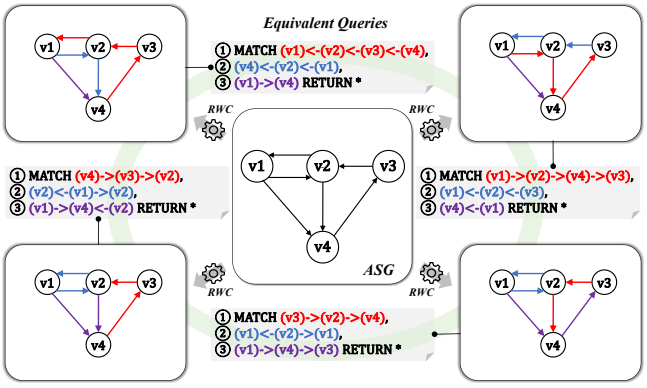


Figure 3: An illustrative example of the process of equivalent Cypher query generation through Random Walk Covering (RWC) on an Abstract Syntax Graph (ASG). Due to the space limit, four of the equivalent queries are presented.

Algorithm 1: Generating Equivalent Queries from ASG via Random Walk Covering.

Data: (G, V, E, D): An ASG G contains vertices within V and edges within E; D: A dict of *vertex* : {*constraints*}, where $D[u]$ represents the set of constraints associated with vertex u.
Result: Q_e: an equivalent query generated from the ASG.

$Q_e \leftarrow$ an empty GDBMS query ;
while $V \neq \emptyset$ ***or*** $E \neq \emptyset$ **do**
  $(P_v, P_e) \leftarrow$ Randomly select one *Path* from G that contains vertices within P_v and edges within P_e ;
  for $e \in P_e$ **do** $E \leftarrow E \setminus e$;
  $P_d \leftarrow$ dict() ;
  for $v \in P_v$ **do**
    $P_d[v] \leftarrow$ Randomly select one subset from $D[v]$;
    $D[v] \leftarrow D[v] \setminus P_d[v]$;
    if $D[v] = \emptyset$ ***and*** *no edge connected with* $v \in E$ **then** $V \leftarrow V \setminus v$;
  end
  $Q_e \leftarrow Q_e \cup$ TransPath2Query(P_v, P_e, P_d) ;
end
return Q_e ;

while a Gremlin query's traversal is made up of a series of linear sub-traversals.

Recognizing this, we can transform an ASG into a sequence of non-overlapping paths, which covers the ASG (*i.e.*, every relationship and constraint within the ASG is represented by these paths exactly once), thereby corresponding to an equivalent GDBMS query. For example, as illustrated in Fig. 3, the ASG can be represented by various path sequences. In these sequences, each path (represented by a different color) matches a subset of the relationships and constraints described by the ASG. Collectively, these paths represent all the information present in the ASG by covering all its vertices and edges. These path sequences are then translated into GDBMS queries, resulting in equivalent queries generated from the ASG.

We propose Random Walk Covering (RWC), a simple and effective algorithm. RWC generates various non-overlapping path

sequences that cover the ASG. Each edge in the ASG belongs to exactly one path. The specifics of this algorithm are detailed in Algorithm 1. The algorithm begins with an empty GDBMS query Q_e (Line 1), and an ASG $G = (V, E, D)$, where V represents the set of vertices, E the set of edges, and D the constraints associated with each vertex. The algorithm proceeds by selecting paths from G in turns, continuing until the entire ASG is covered by the selected paths (Lines 2-12). In each turn, a random path from G is selected, denoted as (P_v, P_e), where P_v and P_e represent the vertices and the edges within the path, respectively (Line 2). Every edge e in the set P_e is then removed from the edge set E, having been covered by the path (Line 3). Simultaneously, we initialize an empty dictionary P_d to represent the constraints carried by the path (Line 4). For every vertex v in the set P_v, a random subset of its constraints is removed from $D[v]$ and assigned to $P_d[v]$ (Lines 7-8). If all edges connected with v and the constraints on v are removed (*i.e.*, $D[v]$ becomes $\emptyset$), we then remove v from the vertex set V, as it and its constraints have already been covered by selected paths (Line 9). Subsequently, the *TransPath2Query()* function is used to translate the path and its associated constraints back into the query language of the GDBMS. The resulting query section is then appended to Q_e (Line 11). This process is repeated until all information linked to G has been covered (i.e., $V = \emptyset$ and $E = \emptyset$), at which point Q_e becomes an equivalent query that aligns with the ASG (Line 2, 12).

In addition, these equivalent queries will produce diverse query plans and traversal strategies for retrieving the desired data. For example, consider the equivalent Cypher queries `MATCH (n1:person) -[]->(n2:book)` and `MATCH (n2:book)<-[]-(n1:person)`. They can derive different query plans in some GDBMS, with the first query retrieving all person nodes first, while the second query retrieves all book nodes first. Although certain GDBMS can optimize these two queries into the same query plan (*i.e.*, by searching the smaller set of nodes first), it becomes challenging to optimize different path sequences into a unified query plan for more complex queries and path sequences. This limitation arises from the inability of GDBMS to fully predict the graph structure and the searching space before query execution. Therefore, RWC on ASGs allows for comprehensive and effective testing of GDBMS by generating and executing equivalent queries with various query plans.

4 IMPLEMENTATION

4.1 Implementation Overview

In this study, we have developed a tool called *GRev* for testing GDBMS, which encompasses three components: *Graph Generator*, *Base Query Generator*, and *Equivalent Query Rewriter*.

The overview of GRev is shown in Fig. 4. Initially, we utilize a graph generator to randomly create a graph schema in the target graph database, containing labels, properties, nodes, and relationships. Following that, we use the base query generator to produce a base query, denoted as Q_{base}. Subsequently, we employ the equivalent query rewriter, which utilizes the aforementioned EQR approach, to rewrite Q_{base}, generating multiple equivalent queries, denoted as $Q_1, Q_2, \ldots,$ and Q_k, where each query maintains the equivalence of Q_{base}. By executing these queries on the target graph database, potential bugs can be uncovered if they result in inconsistent systems behaviors.

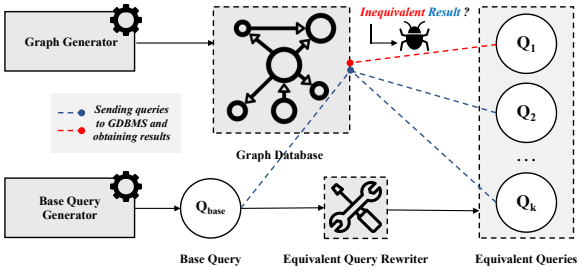


Figure 4: An overview of GRev, which begins with the Graph Generator and Base Query Generator to produce a graph schema and base query, followed by the Equivalent Query Rewriter to generate several equivalent queries. A bug will be detected if these queries yield inconsistent system behaviors.

4.2 Graph Generation

Unlike some testing techniques [13, 15] that are specific to particular data, our EQR approach can be generally applied to any graph. We can simply create a random LPG graph schema on the target graph database by executing the following steps.

- First, create properties and assign them unique names and data types (*e.g.*, `int`, `float`, and `string`).
- Second, create labels for nodes and relationships, with each label corresponding to a set of randomly selected properties.
- Third, create nodes and relationships, assigning them random labels and properties, and then assigning random values to their properties based on their data-types.

4.3 Base Query Generator

To the best of our knowledge, existing tools for testing GDBMS are often sensitive to the base query. For instance, differential testing tools like Grand [44] and GDsmith [13] are limited to using queries that rely on shared functionalities among the tested graph databases. Similarly, GDBMeter [15], a tool based on Query Partitioning [32], requires different validation rules for different queries and may struggle to handle some widely-used clauses such as `DISTINCT`. This is because there can be potential duplicate elements across the sub-queries, and thus aggregating sub-queries with `DISTINCT` clauses may not produce results matching the original query.

However, GRev is designed to be generally applicable to most base queries, with the only exception of the base queries that may produce uncertain results, such as those involving the random `LIMIT` clauses. Given this, we followed the grammar of GDBMS query languages to generate syntactically correct base queries. Furthermore, GRev can directly utilize any existing query generators, such as the generator of GDsmith [13]. We also believe that GRev will become even more effective in testing GDBMS when query language generators with higher coverage and efficiency are developed in the future.

4.4 Equivalent Query Rewriter and Validation Rules

GRev can detect multiple types of bugs (*i.e.*, logic bugs, performance bugs, and exception-related bugs). We employ diverse validation

rules to identify and differentiate between these bug types. After generating the graph schema and the base query, GRev employs the aforementioned EQR approach, as described in Section 3, to generate multiple equivalent queries. For a pair of equivalent queries (Q_a, Q_b), we denote their results as $R(Q_a)$ and $R(Q_b)$, and their execution times as $T(Q_a)$ and $T(Q_b)$ respectively. We will detect the potential bugs using the following validation rules, where E_x is defined as the set of exceptions that GDBMS may return.

Detecting Logic Bugs. Logic bugs represent a notorious category of bug that can lead to incorrect query results without triggering any exceptions or warnings. In GRev, logic bugs can be detected by comparing the results between the two equivalent queries. For a pair of equivalent queries (Q_a, Q_b), if $R(Q_a) \neq R(Q_b)$ and no exceptions are thrown, it will indicate logic bugs in the GDBMS, as two equivalent queries produce different results without throwing any exceptions. Specifically, if $R(Q_a) \neq R(Q_b)$ but exceptions are thrown, it does not suggest logic bugs. Instead, it points toward exception-related bugs, which include unexpected exceptions and missing exceptions.

Detecting Performance Bugs. Performance bugs are also considered significant bugs in GDBMSs as they have a negative impact on query response times. In GRev, performance bugs can be detected by comparing the execution time between the two equivalent queries. For a pair of equivalent queries (Q_a, Q_b), if Equation 1 holds, it will indicate a performance bug in the GDBMS, as two equivalent queries result in a significant difference in their execution times.

Specifically, different from the logic bugs, the inconsistency of the running time between two equivalent queries could stem from execution environmental factors, potentially causing false alarms. To balance false alarms and missed detections, we configured $C_1 = 0.8$ and $C_2 = 1000$ms, indicating that each test case triggering a performance bug must have at least $5\times$ slowdown and a time gap of 1000ms or more.

$$\begin{aligned} \frac{\max(T(Q_a), T(Q_b)) - \min(T(Q_a), T(Q_b))}{\max(T(Q_a), T(Q_b))} &\geq C_1, \\ \max(T(Q_a), T(Q_b)) - \min(T(Q_a), T(Q_b)) &\geq C_2, \\ \text{where } C_1, C_2 \text{ are two constant thresholds.} & \end{aligned} \tag{1}$$

5 EVALUATION

In this section, we address the following research questions to evaluate various important aspects of GRev:

- **RQ1: Detecting previously unknown bugs.** How effective is GRev at detecting previously unknown bugs in mature GDBMS?
- **RQ2: Effectiveness in generating significant test cases.** How effective is GRev in generating test cases that are interpreted into distinct query plans[2], and how many of them trigger bugs?
- **RQ3: Comparison with existing techniques.** How does the effectiveness and generalizability of GRev in GDBMS testing compare to contemporary SOTA approaches?

Table 1: The GDBMS we tested are popular and widely used and representative.

GDBMS	DB-Engine Rank	GitHub Stars	Initial Release
Neo4j	1	11.6k	2007
RedisGraph	4*	1.9k	2018
MemGraph	7	1.5k	2017
NebulaGraph	9	9.2k	2019
TinkerGraph	31	1.9k	2009

5.1 Experimental Setting

Five widely-used and representative GDBMS are incorporated in our evaluation (*i.e.*, Neo4j [26], RedisGraph [31], MemGraph [23], TinkerGraph [40], and NebulaGraph [24]). Table 1 shows the basic information of the GDBMS selected, including DB-Engines Ranking [6][3], GitHub stars, and their initial release dates. Among these, Neo4j, RedisGraph, and MemGraph are the most popular open-source GDBMS that employ Cypher as their query language. TinkerGraph, developed by the creators of Gremlin [35], serves as the foundational library for several Gremlin-based GDBMS such as JanusGraph [14] and OrientDB [34]. Notably, these four GDBMS have undergone rigorous testing by existing works (*e.g.*, GDsmith [13], Grand [44], and GDBMeter [15]). Therefore, any previously unknown bugs we discovered on these databases highlight the proficiency of GRev. Additionally, we tested NebulaGraph, a widely-used GDBMS integrating its own query language, demonstrating the generalizability of GRev across multiple query languages.

We tested the most recent versions of the GDBMS at the time of this work, which were Neo4j 5.6.0, RedisGraph 6.2.6-v7, MemGraph 2.8.0, TinkerGraph 3.6.0, and NebulaGraph 3.5.0. All experiments are conducted on a Linux (Ubuntu 22.04 LTS) workstation, equipped with a 13th Gen Intel(R) Core(TM) i5-13400 and 128 GB of memory.

5.2 Detecting Previously Unknown Bugs

An Overview of Detected Bugs. We detected 22 previously unknown bugs across five GDBMS, with 15 confirmed by developers. In addition to these bugs, 3 reported bugs in Neo4j [26] are already known to developers but not publicly disclosed. These were fixed in the latest Neo4j version and are not considered detected bugs. In addition, one identified Neo4j issue involved developers intentionally withholding an exception for new features. Though this exception is unexpected in both previous and future versions, we do not classify it as a confirmed bug.

Table 2 shows the overall bug statistics. Out of 22 reported bugs, 12 are logic bugs, 3 are performance bugs, 4 are missing exceptions, and 3 are unexpected exceptions. Among these bugs, 15 of them have been fixed or confirmed by the developers, while the status of the remaining 11 bugs is still pending. Upon a detailed investigation of these bugs' manifestations, we further identified 14 of them as *graph-related*. These bugs arise from improper implementation of graph data retrieval in GDBMS (*e.g.*, `MATCH` clause in Cypher),

[2]In GDBMS, distinct query plans mean that two functionally equivalent queries are executed using different strategies, not just differing in their textual representation. .
[3]Statistics as of July 2023, where RedisGraph's rank (4*) includes secondary database models.

Table 2: Summary of Detected Bugs: During our testing of 5 different GDBMS adopting 3 distinct query languages, we discovered a total of 22 previously unknown bugs, 15 already confirmed by the developers. Of the 22 reported bugs, 12 are logic bugs, 3 are performance bugs, 4 are missing exceptions, and 3 are unexpected exceptions.

GDBMS	Query Language	Detected	Confirmed	Logic Bugs	Performance Bugs	Missing Exceptions	Unexpected Exceptions	Graph-related
Neo4j	Cypher	3	2	0	0	2	1	1
RedisGraph	Cypher	6	2	4	1	0	1	4
MemGraph	Cypher	4	3	3	0	1	0	3
TinkerGraph	Gremlin	3	2	2	0	1	0	2
NebulaGraph	nGQL	6	6	3	2	0	1	4
Total		**22**	**15**	**12**	**3**	**4**	**3**	**14**

rather than common functionalities shared with RDBMS (*e.g.*, predicates). Notably, these bugs were rarely detected by the existing techniques, especially for GDBMeter [15], which reused the test oracles designed for RDBMS (*i.e.*, TLP [32]).

Selected Interesting Bugs. To provide an insight into the variety of bugs that GRev is capable of identifying, we present a selection of interesting cases discovered by our approach. For brevity, we present simplified test cases (without showing graph data) for better understanding.

Listing 2: *MemGraph (logic bug, graph-related bug)* - Returning unmatched data when executing `MATCH` clauses after `OPTIONAL MATCH` clauses.

```
(1) MATCH (n0 :L3)<-[r0 :T1]-(n1)
OPTIONAL MATCH (n2 :L2)<-[]-(n1 :L1), (n1 :L1)-[]->(n0)
WITH * MATCH (n0)<-[]-(n1)-[]->(n2) WITH * RETURN *
--{n0 : 19244, n1 : 19230, n2 : 19328, r0: 85207} ✓

(2) MATCH (n0 :L3)<-[r0 :T1]-(n1)
OPTIONAL MATCH (n0)<-[]-(n1 :L1)-[]->(n2 :L2)
WITH * MATCH (n0)<-[]-(n1)-[]->(n2) WITH * RETURN *
--{n0 : 19244, n1 : 19230, n2 : 19328, r0: 85207}
--{n0 : 19317, n1 : 19223, n2 : null, r0: 85206}  🐞
```

Bug 1: As shown in Listing 2, these two queries first identify a specific node and relationship pattern with `MATCH`, then explore additional nodes and relationships that might be connected but are not guaranteed to exist using `OPTIONAL MATCH`, at last use `MATCH` again to get the satisfied node and relationships. When executing the first query, which represents the pattern in the `OPTIONAL MATCH` clause in a single path, MemGraph [23] produces the correct result.[4] However, when executing the second query, which is equivalent to the first one but represents the pattern in the `OPTIONAL MATCH` clause as two separate paths, an extra and unwanted row (*i.e.*, `n0`: 19317, `n1`: 19223, `n2`: null, `r0`: 85206) is returned. This row is invalid because `n2` should match a non-empty node in the last `MATCH` clause. The main cause of this bug is the improper optimization of the `MATCH` clauses following the `OPTIONAL MATCH` clauses in MemGraph. During our continued investigation, the key to detecting this bug involves using ASG to represent a pattern with two paths (shown in **(1)**) as an equivalent form to that with only one path (shown in **(2)**), while additional transformations, such as reversing the edges in the pattern, will still yield identical results. Additionally, during the developer's investigation of this test case, they discovered another bug that results in the logic inconsistency shown in Listing 2. Therefore, we find this bug interesting as it showcases the potential of a single test case generated by GRev to uncover multiple real bugs in GDBMS.

Listing 3: *RedisGraph (logic bug, common bug)* - Returning non-existent data when executing `WHERE` clauses after `OPTIONAL MATCH` clauses.

```
(1) MATCH Pattern(A) WITH n8 OPTIONAL MATCH
(n7:L1)-[r7:T1]->(n10), (n2)<-[r9:T1]-(n12)
RETURN n7.id --{n7.id : 120} ✓

(2) MATCH Pattern(A) WITH n8 OPTIONAL MATCH
(n7:L1)-[r7:T1]->(n10), (n2)<-[r9:T1]-(n12)
WHERE (n7.id = 117) RETURN n7.id --{n7.id : 117}  🐞
```

Bug 2: As shown in Listing 3, the first query showcases a scenario in RedisGraph [31], where a complex pattern, denoted as Pattern(A), is matched first, followed by the usage of an `OPTIONAL MATCH` clause to retrieve additional data.[5] After the traversal process, only one node (`n7.id` : 120) is retrieved by the first query. However, when executing the second query in RedisGraph, which includes an additional `WHERE` clause to filter the node (`n7.id` : 117) at the end of the query, RedisGraph still returns a node with `id` : 117. This result is incorrect since there should be no node with `id` : 117 based on the outcome of the first query, and thus the second query should have returned an empty result. The main cause of this bug is the incorrect implementation of `WHERE` clauses. This demonstrates the versatility of GRev in testing various functionalities beyond graph-related bugs. Note the two queries are not equivalent because they are reduced and slightly modified to better reveal the root cause of the bug in the bug report.

Bug 3: As shown in Listing 4, the execution time of the second query in RedisGraph is approximately 9 × slow down than that of the first query.[6] However, both queries are equivalent; the only difference is the presence of a redundant node (`n0`) in the second

[4] https://github.com/memgraph/memgraph/issues/948

[5] https://github.com/RedisGraph/RedisGraph/issues/3100

[6] https://github.com/RedisGraph/RedisGraph/issues/3091

Listing 4: *RedisGraph (performance bug)* - Performance inconsistency caused by duplicated node in the `MATCH` clauses.

```
(1) MATCH (n0 :L3 :L2), (n1 :L0)
MATCH (n2 :L1)<-[r2 :T4]-(n3 :L3), (n0 :L3)
WITH n2 MATCH (n4 :L3)<-[r3 :T3]-(n1), (n0)
WHERE (n2.k7) RETURN COUNT(*) --{performance : 0.54s} ✓

(2) MATCH (n0 :L3 :L2), (n1 :L0)
MATCH (n2 :L1)<-[r2 :T4]-(n3 :L3), (n0 :L3), (n0)
WITH n2 MATCH (n4 :L3)<-[r3 :T3]-(n1), (n0)
WHERE (n2.k7) RETURN COUNT(*) --{performance : 4.82s} 🐞
```

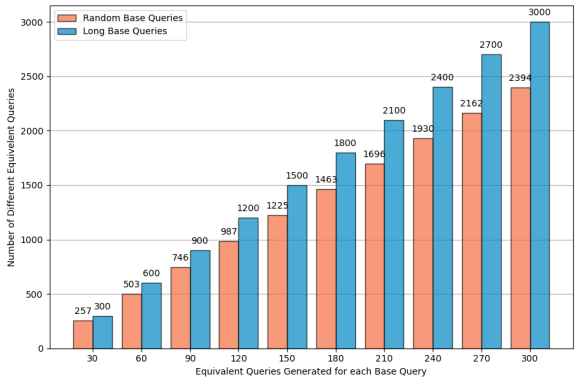


Figure 5: GRev Analysis: How many different equivalent queries can be generated from the 10 base queries within the specified number of generations?

query, and this redundant node causes an inefficient query plan, involving unnecessary scanning and Cartesian product operations for all nodes (`n0`), resulting in significant time overheads. The root cause of this bug lies in the lack of optimization of `MATCH` clauses during query plan generation. This test case highlights the capability of GRev in detecting performance bugs.

5.3 Test Cases Analysis

To investigate the capability of GRev in generating high-quality equivalent queries with diverse query plans, we conducted empirical experiments to answer the following questions:

- **RQ2 (a)**: How diverse are the equivalent queries generated by GRev for each base query?
- **RQ2 (b)**: How diverse are the query plans generated by GRev for each base query?
- **RQ2 (c)**: What proportion of the test cases generated by GRev are bug-triggering?

To address **RQ2 (a)** and **RQ2 (b)**, we generate base queries using a standard query generator of GDsmith [13] for efficient analysis. The reason for using the third-party query generator here is to reduce the bias introduced by the generator. We focused on two groups of base queries:

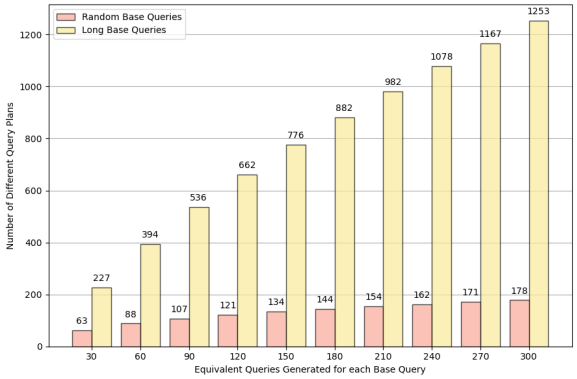


Figure 6: GRev Analysis: How many different query plans can be generated from the 10 base queries within the specified number of generations?

- *Random Base Queries*: This involved randomly selecting 10 base queries from a set of 5,000 queries generated by GDsmith.
- *Long Base Queries*: This involved selecting the top 10 longest base queries from the same set of 5,000 queries.

We then employed our EQR approach to generate 300 equivalent queries for each base query and derived their query plans in RedisGraph 6.2.6-v7. Our analysis involved tracking the number of distinct queries and query plans observed within the set of equivalent queries. We conducted the experiment 50 times, and took the average results presented in Fig. 5 and Fig. 6.

Will GRev fall into generating duplicated queries? As shown in Fig. 5, GRev demonstrates its effectiveness in generating diverse equivalent queries: (**2,394**/**3,000**) unique equivalent for random base queries and (**3,000**/**3,000**) unique equivalent queries for long base queries, which means GRev can keep generating diverse queries. As shown in Fig. 6, GRev is also effective and efficient in generating diverse query plans. Specifically, we use 10 base queries (random or long), ask GRev to generate equivalent queries, and record the number of distinct query plans triggered by the equivalent queries. We can observe that when asking GRev to generate 30 equivalent queries for each base query (300 in total), we can obtain 63 distinct query plans for random base queries and 227 distinct query plans for long base queries. The number of distinct query plans obtained for a base query increases as the number of generated equivalent queries increases. If we generate 3,000 equivalent queries for the 10 random base queries (300 each), we can obtain 178 distinct query plans (the rightmost pink bar in Fig. 6), averaging 17.8 distinct plans per base query. Similarly, we can obtain 1253 distinct query plans (the rightmost yellow bar in Fig. 6), averaging 125.3 distinct query plans per base query. Thus, we believe the equivalent queries generated by GRev can trigger distinct query plans. In contrast, existing techniques based on certain query transformations, such as GDBMeter [15], can only trigger 1 (from the base query) + 3 (from three sub-queries) = 4 distinct query plans. Moreover, differential testing tools like GDBMeter and Grand are restricted to triggering only one query plan in each target GDBMS for a given base query. Consequently, when executing differential testing on

Table 3: Comparing GRev with Existing Techniques

Tool	Approach	Supported Languages	Able to detect shared bugs	Able to test unique features	Insensitive to base query	Detected performance bugs	Detected graph-related bugs
GDsmith	Differential Testing	Cypher	○	○	○	○	●
Grand	Differential Testing	Gremlin	○	○	○	○	○
GDBMeter	Query Partitioning	Cypher, Gremlin	●	●	○	○	○
GRev	Query Rewriting	Cypher, Gremlin, nGQL	●	●	◐	●	●

three GDBMS simultaneously, these tools can only activate three distinct query plans for each base query.

To investigate **RQ2 (c)**, we executed 1,250,000 test cases on RedisGraph [31] within a 12-hour window, where each test case includes one base query and one equivalent query. Among these test cases, (**945**/**1,250,000**) managed to trigger bugs, of which 936 are logic bugs and 9 are performance bugs. This result indicates that on average, GRev is able to identify one bug-triggering test case executing for every 1,322 test case executions. In our practical testing, it takes an average of one minute to identify a bug-triggering test case.

In summary, the analysis of the test cases above demonstrates the high effectiveness and efficiency of GRev in generating high-quality test cases and triggering bugs.

5.4 Analytical Comparison with Existing Techniques

As shown in Table 3, there are three closest-related works in testing GDBMS, *i.e.*, GDsmith [13], Grand [44], and GDBMeter [15], that can detect bugs other than crashes. Specifically, GDsmith and Grand are based on *Differential Testing* [38] while GDBMeter is based on *Query Partitioning* [32] (*i.e.*, partitioning query into multiple equivalent sub-queries). Notably, we can not compare GRev with RDBMS testing tools, such as SQLancer [33]. Although GRev and these tools both test the bugs in database systems, their target databases have fundamental differences, in terms of data structures (*e.g.*, graph model vs. relational model), query mode (*e.g.*, traversals vs. joins), and languages (*e.g.*, Cypher [28], Gremlin [35], and nGQL [27] vs. SQL).

Even for testing the GDBMS that adopt the same query languages, different GDBMS can have their unique design or adopt some shared libraries with other GDBMS. In the first case, these differential testing tools are sensitive to queries or are prone to give false positives. For example, MemGraph [23] and Neo4j [26] will deliberately return different results for the same query `MATCH ()-[r]-()`, and thus GDsmith does not test such queries to avoid false alarms. In the second case, these tools are prone to give false negatives. In addition, GDsmith and Grand face challenges in detecting performance bugs by comparing performance discrepancies, as different target GDBMSs have varying underlying architectures.

Based on Query Partitioning, GDBMeter [15] leverages the test oracle called Ternary Logic Partitioning (TLP) to detect bugs in GDBMS. TLP [32] was initially introduced in testing RDBMS. It involves partitioning a query Q into sub-queries $Q_P, Q_{\neg P}, Q_{\text{null}}$ based on three possible outcomes of the predicate P: true, false, and null. However, this partitioning approach is sensitive to query and query language. For example, where the query includes functionalities such as removing duplicate elements, aggregating the sub-queries may no longer produce an equivalent result to the original query.

Listing 5: An example of GDBMeter's missed bug-triggering test cases of the bug shown in Listing 2, where the aggregation of the results from the three sub-queries still produces the same incorrect result as the original query, which leads to the bug being missed.

```
(TLP : Q_P) MATCH (n0 : L3)<-[r0 : T1]-(n1)
OPTIONAL MATCH (n0)<-[]-(n1 : L1)-[]->(n2 : L2)
WITH * MATCH (n0)<-[]-(n1)-[]->(n2) WITH *
WHERE (ID(n2) > ID(n0)) RETURN *
--{n0 : 19244, n1 : 19230, n2 : 19328, r0: 85207}

(TLP : Q_¬P) MATCH (n0 : L3)<-[r0 : T1]-(n1)
OPTIONAL MATCH (n0)<-[]-(n1 : L1)-[]->(n2 : L2)
WITH * MATCH (n0)<-[]-(n1)-[]->(n2) WITH *
WHERE NOT (ID(n2) > ID(n0)) RETURN * --{}

(TLP : Q_null) MATCH (n0 : L3)<-[r0 : T1]-(n1)
OPTIONAL MATCH (n0)<-[]-(n1 : L1)-[]->(n2 : L2)
WITH * MATCH (n0)<-[]-(n1)-[]->(n2) WITH *
WHERE (ID(n2) > ID(n0)) IS NULL RETURN *
--{n0 : 19317, n1 : 19223, n2 : null, r0: 85206}
```

As a result, GDBMeter is unable to effectively test queries that involve `DISTINCT` clauses in Cypher [28] and `dedup()` clauses in Gremlin [35]. Similarly to Grand [44] and GDsmith [13], GDBMeter also lacks general support for multiple query languages in GDBMS. For example, since Gremlin lacks steps to check for null values in expressions, GDBMeter can only test Gremlin-based GDBMS when working with graphs that do not contain null values. In addition, TLP can only be applied to filter clauses (*e.g.*, `WHERE`), and is not capable of effectively modifying query plans of the traversal process (*e.g.*, `MATCH`), resulting in GDBMeter facing challenges in finding traversal bugs. For example, as shown in Listing 5, the TLP approach fails to generate bug-triggering test cases for the graph-related bugs shown in Listing 2. Specifically, after conducting a thorough investigation of GDBMeter's bug reports, we found that all of the logic bugs were triggered by queries that contain only a simple `MATCH (n)` clause or return a mathematical expression directly. This implies that the root causes of their detected logic bugs lie solely in the implementations of predicates and expressions, with no relation to the graph itself.

5.5 Empirical Comparison with Existing Techniques

To better compare with previous works, we conducted two empirical experiments, each corresponding to code coverage, and bug

Table 4: Graph Databases and Versions of Previous Works Tested

Tool	Graph Database	Versions
GDsmith	Neo4j Community Edition	3.5,4.2,4.3,4.4,4.5
	RedisGraph	2.8
	Memgraph Community Edition	2.4
GDBmeter	Neo4j Community Edition	4.4.8,4.4.9
	RedisGraph	2.8.19
Grand	TinkerGraph	3.4.10

Table 5: Bug Reproducing on Different GDBMS, where *all* indicates that all the versions of database mentioned in Table 4 are tested.

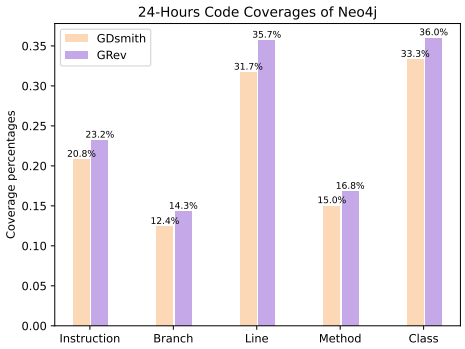
Graph Database	Version	Reproduced
Redis Graph	*all*	**5**/6
Memgraph	*all*	**3**/4
TinkerGrpah	*all*	**1**/3
Total	*NA*	**9**/13

reproduction. As shown in Table 4, we choose those tools that test the shared databases with us.[7]

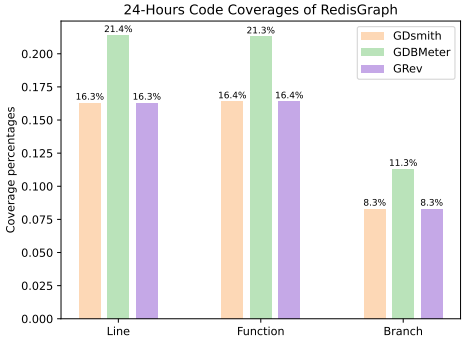
The 24-Hour Testing Coverage Comparison. We compare GRev's coverage with GDBmeter and GDsmith on RedisGraph, and GDsmith on Neo4j.[8] To be fair, GRev uses GDsmith to generate the base queries in this experiment. As shown in Figure 7, the result of testing Neo4j shows that GRev has higher instruction coverage and branch coverage compared to other testing frameworks, which means GRev has the capability to trigger more instructions/branches related to query plan, given a fixed query generator. The result of testing RedisGraph shows that GDBmeter has higher code coverage than others, and GRev yields the same coverage as GDsmith, which indicates that Neo4j may have more optimizations than RedisGraph. We think GDsmith produces certain queries based on the template, which restricts the diversity of queries. On the other hand, the base queries produced in GDBmeter are more diverse, thus having higher coverage. Specifically, we posit that if GRev employs some identical generator as GDBMeter, it could achieve at least the same code coverage as that from GDBMeter in RedisGraph.

Bug Reproduce Comparison. To assess whether other tools can detect the bugs identified by GRev, a straightforward approach is to attempt to reproduce these bugs on the versions of databases tested by those tools. If these successfully reproduced bugs were indeed found by other tools, they would likely have been reported to developers and subsequently fixed. We conducted tests on the GDBMS listed in Table 4. Our focus was on reproducing bugs initially detected in MemGraph, RedisGraph and TinkerGraph across

[7]Grand is excluded in the first two experiments, due to the missing YAML configurations for databases.
[8]GDBmeter can not be compared in Neo4j, since the configuration for Neo4j is missing in GDBmeter.



(a) Code Coverages of Neo4j



(b) Code Coverages RedisGraph

Figure 7: 24-Hours Code Coverages Comparison

all versions examined by existing methodologies. This investigation aimed to determine whether these bugs existed in early versions and were overlooked by prior testing efforts. Note that we do not manually examine bug reports from other tools because they often report complex test cases that trigger bugs without specifying the root cause. It is challenging and time-consuming to manually check whether these reports are identical to the bugs found by GRev.

The results in Table 5 show that 69% (9/13) of the bugs detected by GRev can be reproduced in earlier versions, except for Neo4j. This is attributed to the discovery of new bugs, particularly those related to Neo4j's new features, such as label expression. The existing methods focus on transformation rules for shared functionalities with RDBMS, leaving graph-related bugs in early versions undetected. This empirical evidence underscores the effectiveness of GRev in uncovering bugs that may go unnoticed by other tools.

6 DISCUSSION

6.1 Threats to Validity

First, while GRev tests GDBMS automatically, bugs require manual reduction and reproduction, possibly introducing errors. To mitigate this, three individuals carefully study both the original bugs and the reduced bugs and reach a consensus before reporting them to the developers. Second, given the diversity of GDBMS query languages, testing GRev's universality across all isn't feasible. To address this limitation, we focus on the three most popular query languages based on the DB-Engine Ranking. Third, there is no

well-established performance inconsistency standard in GDBMS for performance bugs. Thus, the thresholds used might lead to false alarms. To mitigate this, we set a time difference threshold to filter out possible false alarms, similar to AMOEBA and Apollo, the existing works in detecting logic issues in RDBMS. In addition, we will automatically reproduce the performance-inconsistent test cases 5 times once it was detected, to ensure they can trigger the bugs stably. As a result, none of our reports of performance issues have been rejected (false alarms).

6.2 Equivalent Queries Generated by RWC

RWC is capable of generating a substantial number of equivalent queries for any given ASG. The quantity of equivalent queries produced by RWC equals or surpasses the number of different ordered path-covering schemes of G. For the path-covering schemes within a given G, a relaxed lower bound can be established by exclusively considering paths of length 1 (*i.e.*, paths collapsing into edges) for covering G. Under this scenario, supposing the number of edges of G is $|E|$, we can create $|E|!$ distinct order of placing edges, where each edge (denoted as <a, b>) can be represented in two forms (*i.e.*, <a, b> and <b, a>), resulting in $2^{|E|}$ various edge representations. Thus, even when we limit the cases to paths in the RWC to single edges, there remain $2^{|E|}|E|!$ diverse equivalent queries (*i.e.*, path-covering schemes) that can be derived from the ASG. For instance, when $|E| = 5$, this lower bound equals 3840.

7 RELATED WORK

Differential Testing. Differential testing is a widely employed technique for bug detection in systems. The fundamental idea behind it [10, 22] is to verify that, for a given input, different systems produce identical execution results. Leveraging this principle, differential testing effectively identifies inconsistencies between systems and finds practical applications in diverse research domains, including binary program analysis [29], desktop hardware evaluation [11], JVM investigations [4, 5], debugger assessments [18], AI systems examination [1, 7–9, 20], object-relational mapping systems [37], etc. Grand [44] and GDsmith [13] represent two established and prominent differential testing tools specifically designed for GDBMS. However, they can not support different query languages, provide test oracles for the single target GDBMS, and detect shared bugs across multiple GDBMS. In this work, our EQR approach addresses these challenges by providing a universal framework ASG and a test oracle based on metamorphic testing.

Metamorphic Testing. Metamorphic testing has been successfully applied in various domains [12, 19, 42, 43]. For example, Orion [16], Athena [17], and Hermes [39] introduce metamorphic testing techniques in compilers such as equivalent dead code mutation and equivalent live code mutation to identify logic bugs within the compiler. Based on metamorphic testing, GDBMeter [15] is the SOTA tool for detecting logic bugs in GDBMS. However, none of the graph-related bugs have been detected by it due to reusing TLP[32], a metamorphic relation for RDBMS. In this work, we propose a novel and more effective metamorphic testing method for GDBMS that can detect graph-related bugs and common bugs shared with RDBMS.

Equivlanet Rewriting in RDBMS Testing. Both AMOEBA[21] and GRev are able to generate equivalent queries to trigger performance bugs in the database. AMOEBA supports 75 mutation rules defined by domain experts, which incurs domain knowledge and extensive effort in rule validation. However, our work does not need extra rules to define the transformation process. Our EQR only depends on the original ASG and uses RWC to automatically generate large numbers of equivalent queries (Section 3.3). This means, for a specific query, AMOEBA can generate at most 75 equivalent queries, while GRev can generate equivalent queries exponentially.

8 CONCLUSION

This paper adapts Equivalent Query Rewriting to GDBMS testing, with two proposed key ideas: (1) *Abstract Syntax Graph (ASG)*, a general concept facilitating the generation of queries with customized properties, and (2) *Random Walk Covering (RWC)*, a simple but effective algorithm for finding paths in an ASG. In addition to equivalence, we believe ASG and RWC can be easily adapted to generate queries with other properties (*e.g.*, queries that should return a subset of the results of the based query), leading to the design of various testing approaches. As a practical realization of these ideas, we developed a tool called GRev. GRev has been proven effective on 5 popular GDBMS, uncovering 22 previously unknown bugs, including logic bugs, performance bugs, and exception issues. In particular, GRev successfully found 14 bugs that are related to improper implementation of graph data retrieval in GDBMS, which is different from existing techniques that mainly report bugs related to similar functionalities in RDBMS. In addition, the GDBMS tested collectively adopt three different query languages, which is the most compared to existing research.

9 DATA AVAILABILITY

Codes for GRev and for reproducing all the experimental results are available at GitHub.[9]

ACKNOWLEDGMENTS

We thank the anonymous ICSE reviewers for their valuable feedback on the earlier draft of this paper. We also want to thank all the GDBMS developers for responding to our bug reports as well as analyzing the bugs we reported. We especially want to thank Wenlin Wu, a NebulaGraph developer, for his insightful feedback on the performance issues and for taking all the reported bugs seriously and quickly. This paper was supported by the National Natural Science Foundation of China (No. 62102340), Shenzhen Science and Technology Program, and Shenzhen Research Institute of Big Data.

REFERENCES

[1] Muhammad Hilmi Asyrofi, Zhou Yang, and David Lo. 2021. Crossasr++: A modular differential testing framework for automatic speech recognition. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1575–1579.

[2] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2060–2071. https://doi.org/10.1109/ICSE48619.2023.00174

[3] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark

[9] https://github.com/CUHK-Shenzhen-SE/GRev

Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 49–60. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[4] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1257–1268.

[5] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99.

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

[7] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2022. Fuzzing deep-learning libraries via large language models. *arXiv preprint arXiv:2212.14834* (2022).

[8] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv preprint arXiv:2304.02014* (2023).

[9] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing deep-learning libraries via automated relational api inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 44–56.

[10] Robert B Evans and Alberto Savoia. 2007. Differential testing: a new approach to change detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*. 549–552.

[11] Alex Groce, Gerard Holzmann, and Rajeev Joshi. 2007. Randomized differential testing as a prelude to formal verification. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 621–631.

[12] Pinjia He, Clara Meister, and Zhendong Su. 2020. Structure-invariant testing for machine translation. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 961–973.

[13] 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)*.

[14] JanusGraph. 2023. *JanusGraph.* https://janusgraph.org/

[15] Matteo Kamm, Manuel Rigger, Chengyu Zhanga, and Zhendong Su. 2023. Testing Graph Database Engines via Query Partitioning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[16] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.

[17] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices* 50, 10 (2015), 386–399.

[18] Daniel Lehmann and Michael Pradel. 2018. Feedback-directed differential testing of interactive debuggers. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 610–620.

[19] Mikael Lindvall, Dharmalingam Ganesan, Ragnar Árdal, and Robert E Wiegand. 2015. Metamorphic model-based testing applied on NASA DAT–An experience report. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 129–138.

[20] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 530–543.

[21] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering*. 225–236.

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

[23] MemGraph. 2023. *MemGraph.* https://memgraph.com/

[24] NebulaGraph. 2023. *NebulaGraph.* https://www.nebula-graph.io/

[25] Neo4j. 2023. *Cypher Features Implemented by Neo4j.* https://neo4j.com/docs/cypher-manual/current/queries/basic/#find-connected-nodes

[26] Neo4j. 2023. *Neo4J.* https://neo4j.com/

[27] nGQL. 2023. *nGQL.* https://docs.nebula-graph.io/1.2.0/manual-EN/1.overview/1.concepts/2.nGQL-overview/

[28] openCypher. 2023. *openCypher.* https://opencypher.org/

[29] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. 2017. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP)*. IEEE, 615–632.

[30] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (San Diego, California, USA) *(SIGMOD '92)*. Association for Computing Machinery, New York, NY, USA, 39–48. https://doi.org/10.1145/130283.130294

[31] Redis. 2023. *RedisGraph.* https://docs.redis.com/latest/stack/deprecated-features/graph/

[32] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 211 (nov 2020), 30 pages. https://doi.org/10.1145/3428279

[33] 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)*, Vol. 20. 667–682.

[34] Daniel Ritter, Luigi Dell'Aquila, Andrii Lomakin, and Emanuele Tagliaferri. 2021. OrientDB: A NoSQL, Open Source MMDMS. In *Proceedings of the The British International Conference on Databases 2021, London, United Kingdom, March 28, 2022 (CEUR Workshop Proceedings, Vol. 3163)*. CEUR-WS.org, 10–19.

[35] Marko A. Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language. *CoRR* abs/1508.03843 (2015). arXiv:1508.03843

[36] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing Database Systems via Differential Query Execution. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2072–2084. https://doi.org/10.1109/ICSE48619.2023.00175

[37] Thodoris Sotiropoulos, Stefanos Chaliasos, Vaggelis Atlidakis, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Data-oriented differential testing of object-relational mapping systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1535–1547.

[38] Rainer Storn and Kenneth Price. 1997. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization* 11, 4 (1997), 341–359.

[39] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*. 849–863.

[40] TigerGraph. 2023. *TigerGraph.* https://www.tigergraph.com

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

[42] Boxi Yu, Yiyan Hu, Qiuyang Mang, Wenhan Hu, and Pinjia He. 2023. Automated Testing and Improvement of Named Entity Recognition Systems. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, San Francisco, CA, USA,) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 883–894. https://doi.org/10.1145/3611643.3616295

[43] Boxi Yu, Zhiqing Zhong, Xinran Qin, Jiayi Yao, Yuancheng Wang, and Pinjia He. 2022. Automated testing of image captioning systems. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 467–479.

[44] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. 2022. Finding Bugs in Gremlin-Based Graph Database Systems via Randomized Differential Testing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 302–313. https://doi.org/10.1145/3533767.3534409