



Testing Graph Databases via Transformations Between Fixed-Length and Variable-Length Queries

Jinxin Gui

State Key Lab for Novel Software
Technology, Nanjing University
Nanjing, China
jinxingui@smail.nju.edu.cn

Yuanhong Lan

State Key Lab for Novel Software
Technology, Nanjing University
Nanjing, China
yhl@smail.nju.edu.cn

Longlong Lu

State Key Lab for Novel Software
Technology, Nanjing University
Nanjing, China
lull.scholar@gmail.com

Yifei Lu*

State Key Lab for Novel Software
Technology, Nanjing University
Nanjing, China
lyf@nju.edu.cn

Minxue Pan*

State Key Lab for Novel Software
Technology, Nanjing University
Nanjing, China
mxp@nju.edu.cn

ABSTRACT

The ability of Graph Database Management Systems (GDBMSs) to efficiently store and query graph data has led to their widespread success. Unlike relational databases, GDBMSs model data as graphs and support expressive queries through graph traversal. Among the core functionalities, fixed-length and variable-length queries are particularly critical, as they underscore fundamental differences from traditional relational query execution. However, the correctness of such queries is notoriously difficult to ensure due to the intricate query semantics and the complexity of underlying optimizations like worst-case optimal joins. This paper presents a novel metamorphic testing approach named PATHTest that exploits result-equivalent transformations between fixed-length and variable-length queries. Specifically, PATHTest incorporates an iterative query generator that supports the generation of diverse and non-empty variable-length queries. During the mutation process, three transformation rules embedded within PATHTest help capture result-equivalent patterns between fixed- and variable-length queries, enhancing its capability to uncover both logical bugs and unexpected errors. Extensive evaluation on PATHTest across seven real-world, widely-used GDBMSs demonstrates the superiority of PATHTest, with 41 previously unknown bugs revealed, among which 24 are logic bugs, and 17 correspond to unexpected errors. To note, all 41 bugs are beyond the reach of the seven existing state-of-the-art testing approaches. By now, 29 of the 41 bugs have been confirmed, with 11 already fixed. Such evaluation results demonstrate the effectiveness and uniqueness of PATHTest in detecting bugs missed by existing testing approaches, contributing to the reliability of modern GDBMSs.

PVLDB Reference Format:

Jinxin Gui, Yuanhong Lan, Longlong Lu, Yifei Lu, and Minxue Pan. Testing Graph Databases via Transformations Between Fixed-Length and Variable-Length Queries. PVLDB, 19(9): 2234 - 2247, 2026.
doi:10.14778/3819518.3819547

*Corresponding authors.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at https://github.com/VLDB2026/VLDB2026_PATHTest.

1 INTRODUCTION

The growing prevalence of highly interconnected data across domains such as social networks [47], fraud detection [19, 68], recommendation systems [34], and knowledge graphs [6, 20] has led to increasing demands for data management systems capable of efficient modeling and querying intricate relational structures. While traditional Relational DataBase Management Systems (RDBMSs) provide a well-established framework for managing structured data, their performance becomes limited when processing multi-hop relationships due to their reliance on tabular representations and join-based query mechanisms. To tackle such issues, Graph DataBase Management Systems (GDBMSs) have emerged as a compelling alternative [62]. By representing data as nodes and edges, GDBMSs provide a more intuitive and efficient abstraction for relationship-centric queries. Moreover, GDBMSs are specifically optimized for graph workloads, incorporating native storage and indexing structures tailored for connectivity queries [7, 30, 31, 37, 41, 50, 74]. Such advantages have contributed to their wide adoption in both industrial and academic contexts to support large-scale graph analytics. In 2024, the GDBMS market is estimated to be worth USD 507.6 million and projected to grow to over USD 2.1 billion by 2030, with a compound annual growth rate of 27.1% [43].

Challenges. However, the increased flexibility and expressiveness introduce new challenges in ensuring the correctness of GDBMSs. Unlike RDBMSs that adhere to well-recognized SQL standards, GDBMSs support diverse graph query languages with different paradigms, attracting increasing focus in both academia [1, 3, 46, 64] and industry [48, 59]. As identified by Angles et al. [1], graph query languages share two core features: graph pattern matching and graph navigation that serve as the underlying mechanisms for **fixed-length** and **variable-length** queries, respectively. Specifically, fixed-length queries specify bounded structural patterns, such

licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819547

as $(n0) - [] \rightarrow (n1)$, and precisely describe the graph structure to be matched. They are typically evaluated through adjacency-based pattern enumeration. In contrast, variable-length queries, such as $(n0) - [*] \rightarrow (n1)$, aim to retrieve paths of arbitrary length, providing a more expressive way to formulate path-oriented queries. However, such increased expressiveness results in a higher computational cost, as the number of candidate paths grows rapidly with the increase in path length. To mitigate this cost, many optimizations have been proposed, such as worst-case optimal joins for variable-length queries [37] and reachability-based techniques specialized for label-constrained traversals (e.g., $-[*] \rightarrow$) [49, 50]. Such optimizations further complicate graph query processing, significantly challenging the correctness assurance of GDBMSs.

A representative example is the logic bug in AgensGraph [65], reported as issue (#728) [12]. Specifically, AgensGraph executes variable-length queries, such as $(n0) - [*1..3] \rightarrow (n1)$ by internally constructing joins over multiple intermediate path objects, each representing the result of a sub-query on the graph’s topology. To mitigate path explosion, AgensGraph introduces several engineering optimizations, among which path objects are invisible unless explicitly bound to variables in the query, rendering several intermediate sub-query objects invisible in the query context. This incorrect optimization, however, introduces a bug where the variable-length query yields an empty result, even though the corresponding fixed-length query produces correct results.

In recent years, testing approaches for GDBMSs have achieved substantial progress, with differential testing [21, 72] and metamorphic testing [25, 29, 39, 42, 70, 73] emerging as the most prominent approaches in GDBMS testing. However, our experimental evaluation reveals that *existing approaches cannot effectively identify bugs related to variable-length queries*. Specifically, differential testing detects inconsistencies by executing the same queries on multiple GDBMSs and comparing the results. Such approaches are inherently limited by the substantial syntactic and semantic divergence among the implementations of different GDBMSs. Metamorphic testing, on the other hand, attempts to verify the correctness of a single GDBMS by examining whether outputs adhere to predefined metamorphic relations under systematic input transformations. However, existing metamorphic testing approaches predominantly target fixed-length queries, while largely neglecting variable-length queries. In detail, State-Of-The-Art (SOTA) testing approaches, such as GDBMeter [29], GraphGenie [25], and Grasp [39] have their transformation rules designed in a coarse-grained manner, focusing on structural rewrites such as reversing edge directions (e.g., transforming $() - [*] \rightarrow ()$ into $() \leftarrow [*] - ()$). While such transformations may alter the execution plan, they leave the underlying variable-length queries’ semantics unchanged. The narrow coverage of variable-length query semantics in these approaches renders them inadequate for detecting bugs unique to variable-length queries. A notable exception is Gamera [73], which performs testing targeting graph-native structures, thereby moving beyond query-syntax transformations. However, its transformation rules depend on reachability relations between nodes and also keep the semantics of variable-length queries unchanged. Our evaluation also indicates that Gamera overlooks quite a number of bugs unique to variable-length queries. Collectively, these factors help explain why the

aforementioned example bug persisted for over two years, until our approach provided test cases that enabled its reproduction.

Approach. To address the aforementioned challenges, we propose a novel metamorphic testing approach named PATHTest. Our key insight is that, while a single fixed-length query cannot capture the full semantics of a variable-length query, carefully combining multiple fixed-length queries can collectively reproduce the results of variable-length queries with bounded lengths. This insight enables two complementary transformations with *result equivalence*: fixed-length queries can be *folded* into variable-length counterparts by merging explicit edge sequences into a variable-length path expression, while variable-length queries can be *partitioned* into sets of fixed-length queries by imposing explicit path length bounds. *Result-equivalent queries* denote that while queries may vary in semantics with different query plans and intermediates, their result sets are equivalent. For example, the variable-length query $(n0) - [*1..2] \rightarrow (n1)$ can be partitioned into two fixed-length queries, i.e., $(n0) - [] \rightarrow (n1)$ and $(n0) - [] \rightarrow () - [] \rightarrow (n1)$, and vice versa via folding, with results remaining consistent. Such result equivalence allows fixed-length and variable-length queries to serve as mutual testing oracles, thus making correctness evaluations of variable-length queries feasible and enabling more effective testing for fixed-length queries with more expressive references.

We implement PATHTest as the *first* query generator supporting variable-length queries, which also maintains the crucial non-empty property [21]. Underpinned by theoretical guarantees, we designed three fine-grained transformation rules based on the folding and partitioning operations, namely **Variable-Length Pattern Folding (VPF)**, **In-Clause Query Partition (IQP)**, and **Cross-Clause Query Partition (CQP)**. As indicated by their names: ① VPF generalizes rigid fixed-length queries into bounded variable-length forms with explicit constraints, unlocking more flexible execution plans and enabling tailored transformations based on system-specific optimization features. ② IQP rewrites a variable-length clause into multiple fixed-length clauses merged within a single subquery, preserving shared query context while stimulating internal query variation. ③ CQP splits the query into separate branches by selecting a variable-length pattern, replacing it with a fixed-length pattern in each branch, and independently transforming the suffix, thereby enabling isolated paths that trigger diverse planner behaviors. Together, these transformation rules preserve result equivalence while inducing diverse execution behaviors, enabling systematic testing of variable-length queries and substantially extending the testing scope of fixed-length queries in GDBMSs.

Evaluations. We evaluated PATHTest on seven famous GDBMSs: Neo4j [55], MemGraph [45], AgensGraph [65], NebulaGraph [22], JanusGraph [10], HugeGraph [9], and TinkerGraph [60], across two dominant graph query languages, Cypher [56] and Gremlin [59]. In total, PATHTest uncovered 41 previously unknown bugs, including 24 logic bugs and 17 unexpected errors. By now, 29 out of the 41 bugs have been confirmed by the developers, among which 11 have already been fixed. Furthermore, we also conducted a comprehensive comparison between PATHTest and seven SOTA testing approaches: GDSmith [21], GDBMeter [29], GRev [42], GraphGenie [25], Gamera [73], GraspDB [39], and Grand [72]. Experimental results show that PATHTest not only effectively detects bugs with

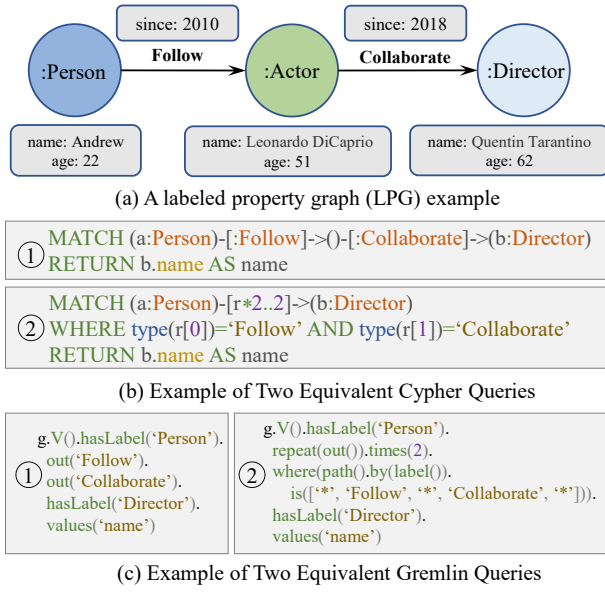


Figure 1: A labeled property graph (LPG) example

zero false-positive rate, but also identifies long-standing issues, including logic bugs that have persisted for over six years. These findings highlight the practical value of our approach as well as its capability to excavate bugs that were missed by prior approaches.

Contributions. This paper makes the following key contributions:

- We propose PATHTest, a novel metamorphic testing approach for detecting logic bugs in GDBMSs by exploiting result equivalence between fixed-length and variable-length queries through a variable-length-pattern-oriented query generator and three fine-grained transformation rules that ensure query diversity.
- We implement PATHTest as a practical tool that automatically generates valid fixed-length and variable-length queries, enabling scalable and effective testing on real-world GDBMSs.
- We conduct a large-scale evaluation of PATHTest on seven representative GDBMSs, demonstrating its effectiveness and complementarity to seven state-of-the-art testing approaches, with 41 previously unknown bugs discovered. So far, 29 of the 41 bugs have been confirmed by developers, with 11 already resolved.

2 BACKGROUND

2.1 Label Property Graph in GDBMSs

Modern GDBMSs typically use the Labeled Property Graph (LPG) model to represent graph data [22, 45, 55, 65]. In this model, the graph consists of a set of *nodes* and a set of directed *edges* between nodes (also referred to as *relationships*). Each *node* can be annotated with one or more *labels* describing its group, while each *edge* is associated with exactly one *label* indicating its type. Both nodes and edges can carry arbitrary *properties* as key-value pairs. For example, as shown in Figure 1(a), the LPG contains three nodes labeled Person, Actor, and Director, and two directed edges labeled Follow and Collaborate. Each node carries two properties: name and age, and each edge has one property since.

2.2 Graph Query Languages

Unlike RDBMSs with SQL that follow well-recognized standards, there exist diverse query languages in GDBMSs [56, 57, 59, 63]. Among these languages, Cypher [56] and Gremlin [59] are the two most recognized and predominant, and are therefore selected as representative languages in this work. Both Cypher and Gremlin support two fundamental graph querying functionalities, namely graph patterns and navigational expressions [1], which correspond to fixed-length path and variable-length path queries, respectively.

Cypher. Cypher is a declarative graph query language for GDBMSs, offering a pattern-matching syntax to query nodes and relationships. Queries in Cypher use SQL-like clauses such as MATCH, WHERE, and RETURN to specify patterns and retrieve data. Query ① in Figure 1(b) shows a fixed-length query that contains two clauses: MATCH and RETURN. The MATCH clause specifies a graph pattern to match, in which a Person node is connected via a Follow edge to an intermediate node and then via a Collaborate edge to a Director node. The RETURN clause specifies which data to return, in this case, the name of the matched Director node. Query ② in Figure 1(c) demonstrates a result-equivalent variable-length query, in which the path is represented using a single variable-length pattern with the * operator to specify the path length, while the edge labels are enforced through type-based predicates in the WHERE clause.

Gremlin. Gremlin is a functional traversal language for GDBMSs, in which queries are expressed as sequences of atomic steps that traverse, filter, or aggregate elements of the graph in a streaming fashion. As an example, the query ① in Figure 1(c) shows a fixed-length traversal using chained steps. This query first selects Person nodes, then traverses Follow and Collaborate edges via consecutive out() steps, and finally filters the result to nodes labeled Director before returning their name values. In contrast, the query ② in Figure 1(c) demonstrates a result-equivalent variable-length traversal, where the entire path is captured using the repeat step with a length constraint via times, and the original edge labels are enforced through a where predicate. These two Gremlin queries are result-equivalent to the Cypher examples in Figure 1(b).

All four queries in Figure 1 are result-equivalent: when executed on the example graph in Figure 1(a), they produce the same result, namely the director *Quentin Tarantino*. For presentation clarity, we primarily use Cypher to describe our approach in the subsequent sections; the corresponding transformations and implementations for Gremlin are later discussed separately.

3 APPROACH

3.1 Result Equivalence of Transformations

Before detailing our approach, we introduce the theoretical foundation of PATHTest to provide a clear understanding of how folding and partitioning establish result equivalence between fixed-length and variable-length queries.

Roughly, we formalize path patterns via Kleene algebra [32, 35, 36], $\mathcal{K} = (Lab, |, \cdot, *, 0, 1)$, where a path can be modeled as an edge sequence. In this formulation: ① *Lab* is the set of edge labels, where we introduce a special symbol *any* $\in Lab$ serving as a wildcard matching any edge label; ② operator $|$ denotes alternation;

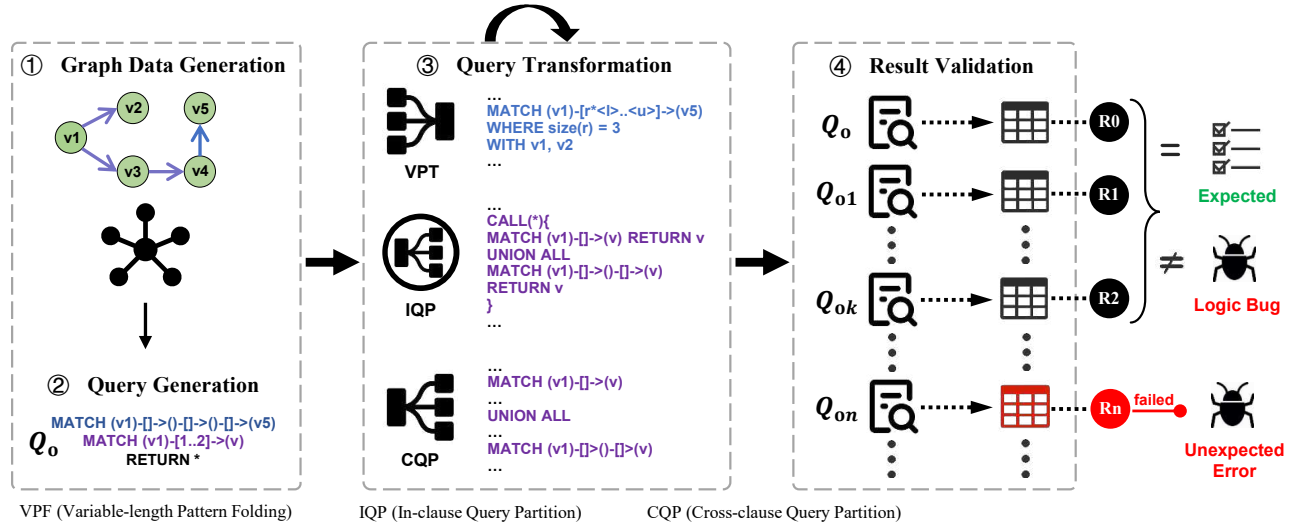


Figure 2: Overview of PATHTest

③ operator $\cdot$ denotes the concatenation of edges; ④ operator $*$ denotes unbounded-repetition; ⑤ constant 0 denotes empty language, matches no paths; ⑥ constant 1 denotes empty string, matches the trivial path of length zero. In addition, Cypher supports variable-length path queries, such as $[:\text{Follow}^*1..u]$ ($0 \leq l \leq u$) to match path length ranging from l to u . To reflect such property, we extend the standard syntax of Kleene algebra with notation $\{l, u\}$ to denote the bounded-repetition within the interval $[l, u]$.

Moreover, we can uniformly represent both fixed-length and variable-length patterns in the queries as:

$$P = x \xrightarrow{\alpha} y, \quad (1)$$

where α denotes the path expression from x to y . We use $\pi \models \alpha$ to denote a concrete path π that matches α , and define $|\pi|$ as the number of edges in π . Accordingly, queries, including fixed-length patterns (e.g., *Follow*), bounded-repetition (e.g., *Follow* $\{2, 3\}$), and unbounded-repetition (e.g., *Follow* $\{2, 3\}$), are well-expressed. For example, the semantics of the pattern in Figure 1(b) can be represented as $x \xrightarrow{\text{Follow} \cdot \text{Collaborate}} y$, which matches all paths that satisfy the "collaborator of a follower" relationship. The proposed transformations, folding (fixed-length patterns can be folded into equivalent bounded-repetition ones using length constraints) and partitioning (bounded-repetition patterns can be partitioned into unions of fixed-length ones), are further discussed below.

For any edge label $L \in \text{Lab}$, $k \geq 1$, bounds $0 \leq l \leq k \leq u$, and $L^{\{l, u\}} = \bigcup_{i=l}^u L^i$, the transformations preserve query semantics.

Folding Equivalence:

$$x \xrightarrow{\overbrace{L \cdot L \cdot \dots \cdot L}^{k \text{ times}}} y \equiv x \xrightarrow{L^{\{l, u\}}} y \text{ with } |\pi| = k, \quad (2)$$

where π is a concrete path matched by $L^{\{l, u\}}$. With the length constraint $|\pi| = k$ matching L^k , the variable-length pattern is identical to the fixed-length pattern.

Partitioning Equivalence:

$$x \xrightarrow{L^{\{l, u\}}} y \equiv \bigcup_{i=l}^u x \xrightarrow{L^i} y. \quad (3)$$

Since the individual path sets of different lengths cannot be identical, the union preserves all matching results without duplication.

Consequently, with such equivalences, any pattern P_1 transformed into P_2 under our proposed transformations matches exactly the same set of paths, building the foundation of PATHTest.

3.2 Approach Overview

Figure 2 presents an overview of PATHTest. PATHTest follows an iterative testing paradigm, with each iteration consisting of four stages. In stages ① and ②, PATHTest constructs a graph database and generates a base query over the database. Next, in stage ③, the base query is transformed into result-equivalent variants to enable metamorphic testing. Eventually, in stage ④, the original query and its transformed variants are executed on the target GDBMSs, followed by an analysis of the execution results. During execution, PATHTest reports two types of potential bugs: (1) *unexpected errors*, including runtime exceptions, system crashes, and internal errors; and (2) *logic bugs*, reported when result-equivalent queries produce inconsistent results. After result validation, PATHTest proceeds to the next iteration by either generating a new query on the current database or constructing a new database, depending on whether the query budget has been exhausted.

3.3 Graph Data and Query Generation

Our graph data and query generation build upon GDSmith [21], a SOTA differential testing framework for Cypher-based GDBMSs. Currently, GDSmith exhibits two primary limitations in practice: ① Although supporting schema-based graph generation, GDSmith lacks compatibility with certain schema-enforced GDBMSs such as NebulaGraph [22]. ② GDSmith only generates fixed-length patterns, without the support of variable-length patterns. To address

such limitations, we leverage a *schema metamodel* to enable schema-aware graph database support and extend query generation with variable-length path capabilities.

Schema Metamodel Generator. GDSmith uniformly models node and edge labels as tuples $(l, \mathcal{P})$, where l denotes the label name and $\mathcal{P}$ is a set of properties. Without explicitly encoding structural constraints on edges, such as the required source and target node labels of an edge type, this model is incompatible with schema-enforced GDBMSs that mandate strict edge schemas, e.g., explicitly declared source and target node labels [22, 58]. To address this limitation, we extend the schema metamodel. The representation of node labels $(l, \mathcal{P})$ remains unchanged, while an edge label is modeled as a quadruple $(l, \mathcal{P}, l_{in}, l_{out})$, where l_{in} and l_{out} denote the labels of the source and target nodes, respectively.

This enhancement enables PATHTest to accurately model and generate schema-compliant property graphs. For example, in NebulaGraph [22], its query language NGQL [61] requires users to explicitly declare **TAG** (i.e., node label) and **EDGE** types at the schema level, which correspond to node label schemas and edge label schemas, respectively. Moreover, each **EDGE** type must specify its source and target **TAG** types before any data can be inserted. Based on our extended metamodel, PATHTest first generates the corresponding CREATE TAG and CREATE EDGE statements from the node and edge label specifications, and then populates the graph using INSERT statements for node and edge instances, ensuring schema compliance and compatibility with the target system.

Variable-length Query Generation (VQG). To generate variable-length queries while retaining effective bug detection, it is essential to preserve the non-empty specificity, a core principle underlying GDSmith’s query generation. Following GDSmith, PATHTest first generates skeletons with database keywords, then fills the skeletons with patterns by iteratively selecting nodes and edges from the generated graph to construct valid paths. This practice helps PATHTest to generate queries mostly with non-empty results. Based on such constraints, we further develop our variable-length query generation component, VQG. Specifically, within the selected subset of nodes and edges, PATHTest identifies sequences of adjacent edges that share the same label, denoted as $L_G \in Lab$, with the maximum number of such adjacent edges being $k \in \mathbb{N}^+$. These sequences correspond to a basic fixed-length pattern of length k . Building upon this, the initial variable-length pattern generated can be constructed as $L_G^{\{k, k\}}$. This serves as a marginal variable-length form that exactly captures the corresponding fixed-length pattern. After that, a series of mutation operations are *randomly* applied to broaden this initial pattern, including reducing the lower bound to a figure in the interval $[0, k)$, increasing the upper bound to a figure in $(k, +\infty)$, and relaxing the label constraint from L_G to *any*, etc. All such mutations are designed to enhance the diversity of generated variable-length path patterns while preserving non-empty specificity, thereby improving the overall effectiveness of PATHTest in uncovering potential bugs during testing.

3.4 Query Transformation

Algorithm 1 illustrates how we traverse a generated query to construct a mutated query. This algorithm takes two inputs: an Abstract Syntax Tree (AST) T representing the structure of the original

Algorithm 1: Pattern-level Query Transformation

Input: AST T ; clause set C
Output: Q_m : an mutated query generated from the AST.

```

1 Function Traversal( $T, C$ ):
2    $Q_m \leftarrow$  an empty GDBMS query;
3   for  $i \leftarrow 0; i < C.size; i++$  do
4      $c \leftarrow C[i]$ ;
5     if  $c$  NOT readingStatement then
6       add  $c$  to  $Q_m$ ;
7     end
8     else if  $c$  has varLenPat  $\wedge$  RandomBoolean() then
9        $C_p \leftarrow$  VarLenthPartition( $T, c$ );
10      if FlipCoin() then
11        format  $C_p$  into subquery  $c'$ ;
12        add  $c'$  to  $Q_m$ ;
13      end
14      else
15         $Q_p \leftarrow$  an empty GDBMS query List;
16        foreach  $c'$  in  $C_p$  do
17           $Q_{pre} \leftarrow Q_m \cup c'$ ;
18           $Q_{suf} \leftarrow$  Traversal( $T, C[i:]$ );
19           $Q_{full} \leftarrow$  combine  $Q_{pre}$  and  $Q_{suf}$ ;
20          append  $Q_{full}$  to  $Q_p$ ;
21        end
22         $Q_m \leftarrow$  union of all queries in  $Q_p$ ;
23        return  $Q_m$ ;
24      end
25    end
26    else if  $c$  has anonPath  $\wedge$  RandomBoolean() then
27       $c' \leftarrow Q_m \cup$  VarLengthFold( $T, c$ );
28      add  $c'$  to  $Q_m$ ;
29    end
30  end
31  return  $Q_m$ ;

```

query, and a clause set C containing its constituent clauses (e.g., MATCH, WHERE, WITH, RETURN, etc.). The algorithm begins with an empty query Q_m (line 2). It then iterates through each clause in C , applying clause-specific rewrites when applicable (lines 3–30). For each clause, it first checks whether it is read-related (e.g., MATCH, OPTIONAL MATCH) capable of pattern analysis. Non-read clauses (e.g., WITH, UNWIND) are appended to Q_m as-is (lines 5–7).

When variable-length patterns are identified, the transformation is applied with a certain probability. If so, the pattern is partitioned into a set of fixed-length subpatterns according to Equation 3, each of which is converted into a separate clause (lines 8–25). We denote the resulting clause set as C_p (line 9). As illustrated in step ③ of Figure 2, a clause such as $(v1)-[*1..2]->(v)$ can be decomposed into two fixed-length clauses: MATCH $(v1)-[]->(v)$ and MATCH $(v1)-[]->()-[]->(v)$. To handle these partitioned clauses, the algorithm randomly selects one of two transformation strategies. The first, referred to as **In-clause Query Partition** (the middle part of step ③ in Figure 2), formats C_p into a single subquery by

combining all clauses using UNION ALL according to Equation 3, and appends it to the current mutated query Q_m (lines 11–12). The second strategy, called **Cross-clause Query Partition** (the lower part of step ③ in Figure 2), constructs multiple complete query branches from the individual partitioned clauses (lines 15–23). Specifically, the algorithm iterates over each partitioned clause $c' \in C_p$. For each c' , it appends c' to the current mutated query Q_m to form a prefix query Q_{pre} (lines 17–18). The remaining clauses $C[i:]$ are then independently and recursively traversed and transformed to generate a set of suffix queries Q_{suf} . Each combination of Q_{pre} and a suffix query in Q_{suf} forms a complete query Q_{full} , which is collected into the query list Q_p (lines 19–20). After all branches are generated, the queries in Q_p are combined using UNION ALL according to Equation 3 to construct the final mutated query Q_m (line 22), which is then returned (line 23).

For fixed-length patterns, the transformation is also applied with a certain probability. If so, we apply the transformation called **Variable-length Pattern Folding**, which folds the path into a variable-length expression to generate a mutated clause (lines 26–29). As shown in the first clause of Q_o in Figure 2, a fixed-length pattern of length three (i.e., $(v1) \rightarrow () \rightarrow () \rightarrow (v5)$) can be folded into a variable-length pattern such as $(v1) \rightarrow [*3..5] \rightarrow (v5)$. We then apply a WHERE clause to filter paths of length exactly three, which, according to Equation 2, ensures result equivalence with the original fixed-length pattern. After all clauses have been processed through one complete traversal, a new query with a fundamentally different path traversal strategy is obtained (line 31).

While each transformation strategy individually guarantees result equivalence between the input and output queries, different strategies yield structurally distinct queries. Randomly selecting among these strategies allows PATHTest to exercise more diverse planning and optimization behaviors in GDBMSs. Subsequently, we further elaborate on these three important strategies.

In-clause Query Partition (IQP). For a variable-length pattern, IQP first splits it into fixed-length clauses and then restructures them into a single subquery, which can be described as:

$$x \xrightarrow{L^{(l,u)}} y \implies \bigcup_{i=l}^u x \xrightarrow{L^i} y, \quad (4)$$

According to Equation 3, this partitioning preserves query semantics by enumerating all possible path lengths between the specified bounds. As illustrated in the middle part of step ③ in Figure 2, to preserve result equivalence with the original clause, the subquery first uses CALL (*) to import the intermediate results from the preceding clauses. This ensures that any nodes or relationships referenced in the partitioned clauses are properly bound. Additionally, we maintain a set of currently live identifiers. Since graph databases typically disallow returning identifiers already defined in the outer scope, the subquery only returns newly introduced identifiers (e.g., the variable v). A minimal RETURN NULL clause is inserted if no new identifiers are introduced. The resulting subquery serves as a semantically valid and structurally complex unit, thus enabling more comprehensive testing of subquery-related behaviors.

Cross-clause Query Partition (CQP). From fixed-length clauses derived from a variable-length pattern, CQP constructs separate queries and then combines them using a UNION ALL operation to

produce a single compound query. To ensure result equivalence with the original query, each new query branch retains all preceding clauses as a shared prefix. This transformation can be described as:

$$Q_o = Q_{pre} \circ x \xrightarrow{L^{(l,u)}} y \circ Q_{suf} \implies \bigcup_{i=l}^u Q_{pre} \circ x \xrightarrow{L^i} y \circ Q_{suf}^{(i)}, \quad (5)$$

where Q_{pre} and Q_{suf} denote the shared prefix and the remaining suffix of the original query; each $Q_{suf}^{(i)}$ is also recursively rewritten to match the path length i . Notably, since each clause can be transformed independently, while different transformations can be further applied on different partitioned branches at later iterations, the final queries become changeful and diverse with result equivalence, thus triggering various query functions with different optimizations and enabling the exposure of latent logic bugs.

To date, though our demonstrations of ICP and CQP focus primarily on bounded paths for clarity, nevertheless, both transformation rules are also applicable to unbounded paths. Specifically, for an unbounded path such as $[*2..]$, we simulate them by splitting out the prefix lengths (e.g., 2, 3) to fix-length subqueries and union them with a variable-length pattern of the remaining unbounded paths (e.g., $[*4..]$). Guided by empirical observations across representative GDBMSs showing that additional fixed-length subqueries do not yield more diverse query plans but incur unnecessary performance overhead, we use the two smallest prefix lengths to balance query-plan coverage and execution efficiency.

Variable-length Pattern Folding (VPF). This transformation strategy folds a fixed-length path pattern into a variable-length pattern with explicit length constraints, described as:

$$x \xrightarrow{\overbrace{L \cdot L \cdot \dots \cdot L}^{k \text{ times}}} y \implies x \xrightarrow{L^{(l,u)}} y \text{ with } |\pi| = k. \quad (6)$$

As illustrated at the top of step ③ in Figure 2, the folded path is assigned randomly generated lower and upper bounds l and u ($l \leq k \leq u$), where k is the original path length. In practice, result equivalence is preserved by introducing a temporary path variable r to bind the variable-length path, enforcing the length constraint in the WHERE clause, and subsequently eliminating r via the WITH clause to prevent propagation. Although the transformation may lead to different query plans, it does not affect the query results. In particular, setting $u = \infty$ yields unbounded repetition, which corresponds to omitting the upper bound in Cypher.

To enhance execution diversity and stress engine-specific optimizations, our strategy extends folded paths by leveraging system-specific features of individual GDBMSs. For example, in Neo4j, a folded pattern such as $(v1) \rightarrow [*3..5] \rightarrow (v5)$ can be rewritten as ALL SHORTEST $(v1) \rightarrow [*3..5] \rightarrow (v5)$, emphasizing shortest-path evaluation. In MemGraph, traversal algorithms can be explicitly specified for patterns (e.g., $(v1) \rightarrow [*BFS \ 3..4] \rightarrow (v5)$), instructing the engine to perform breadth-first search instead of the default depth-first. These forms are syntactically valid and result-equivalent to the originals, enabling the engine to generate more diverse execution plans without altering query results.

Heterogeneous-Label Generalization. For patterns, primarily VPF, involving heterogeneous edge labels or constraints on intermediate nodes, folding a fixed-length path into a variable-length one by directly dropping label constraints and variable bindings

may undermine their result equivalence. To address this issue, our PATHTest adopts a widen-then-constrain strategy, in which the original fixed-length patterns are first folded into a variable-length path p , whose edge label is constrained to the intersection of the labels of the original patterns. This widening step aims to construct a narrow superset of the original patterns, since a smaller superset reduces the number of irrelevant candidate paths considered during matching, thereby improving overall efficiency. In addition, only when the label intersection is empty is the folded path left unconstrained, i.e., its edge label is set to any. Subsequently, constrained by reintroducing the previous constraints from original patterns as predicates over p , we thereby filter out spurious matches and restore the original semantics. For example, consider the pattern $(n1) <- [r1:T0] - (n2:L1) <- [r2:T0:T1] - (n3)$, where adjacent edges impose heterogeneous label constraints, and the intermediate node is label-restricted. We first fold the pattern into a variable-length path using edge-label intersection and bind it to a path variable $p0 = (n1) <- [r:T0*1..2] - (n3)$. The remaining constraints are then enforced as predicates over $p0$ in the WHERE clause. Specifically, constraints on intermediate nodes are preserved by checking the labels at the corresponding path positions, e.g., $ALL(x \text{ IN } ['L1'] \text{ WHERE } x \text{ IN } labels(nodes(p0)[1]))$. Edge labels not covered by the intersection are handled analogously via additional predicates. Finally, to preserve the variable bindings of the original query, we use the path variable to rebind intermediate nodes and relationships to their original variables, e.g., $WITH \ n1, \ r[0] \text{ AS } r1, \ nodes(p0)[1] \text{ AS } n2, \ r[1] \text{ AS } r2, \ n3$. While most property graph databases associate a single label with each edge in practice, multi-label edges are used here to demonstrate how our approach handles both intersecting and non-intersecting label constraints within a unified framework more clearly.

3.5 Implementation

We implemented a prototype of PATHTest for the two most widely used and recognized graph query languages, Cypher [56] and Gremlin [59]. Notably, our approach can also be easily generalized to other modern graph query languages, such as GSQL [57] and GQL [24], with slight engineering effort. For Cypher, our implementation is on top of GDSmith [21] with over 10K lines of Java code. For Gremlin, we first extended Grand’s query generation framework to support variable-length path queries and then followed Algorithm 1 to integrate the same three transformation strategies. Specifically, VPF and CQP were directly implemented in Gremlin, whereas IQP was implemented based on not subqueries but match step, as Gremlin does not support subqueries and match provides similar functionality for presenting inner traversals. Since our implementation directly reuses Grand’s abstract Gremlin traversal model, it requires relatively much lower engineering effort of around 1K lines of code to implement our PATHTest.

4 EVALUATION

Our evaluation is guided by the following Research Questions (RQs):

- **RQ1: Real-world Applicability.** Can PATHTest identify logic bugs in real-world GDBMSs?
- **RQ2: Ablation Evaluation.** How does each component of PATHTest contribute to the overall effectiveness?

Table 1: Information of The Tested GDBMS

GDBMS	Rank	Stars	Year	LoC (K)	Version
Neo4j	1	15.3k	2007	1,582	5.26.4
Memgraph	10	3.4k	2017	334	3.1.1
AgensGraph	31	1.4k	2016	1,681	2.15.0
NebulaGraph	9	11.8k	2019	345	3.8.0
JanusGraph	11	5.7k	2016	155	1.1.0
HugeGraph	28	2.9k	2016	272	1.7.0
TinkerGraph	38	2.1k	2009	1013	3.8.0

- **RQ3: Comparison Evaluation.** How effective is PATHTest compared with the state-of-the-art GDBMSs testing approaches?
- Benchmark GDBMSs.** Seven popular real-world GDBMSs are selected to evaluate the effectiveness of PATHTest on their latest versions: four Cypher-supporting GDBMSs (Neo4j [55], MemGraph [45], AgensGraph [65], and NebulaGraph [22]) and three Gremlin-based systems (JanusGraph [10], HugeGraph [9], and TinkerGraph [60]). Table 1 summarizes their key statistics, including GDB-Engine rankings [40], GitHub stars, Lines Of Code (LOC, measured via `cloc` [8]), and the tested versions. Among the Cypher-supporting GDBMSs, Neo4j [55] is the most prominent, consistently ranking first among GDB-Engines and widely adopted by both industry and the open-source community. MemGraph [45] features an in-memory architecture designed to support both transactional and analytical workloads. AgensGraph [65], built on PostgreSQL [17], integrates relational and graph data models for complex enterprise applications. We also included NebulaGraph [22], which primarily uses its own query language, NGQL [61], but additionally supports OpenCypher [48] for querying. On the Gremlin side, JanusGraph [10] is a highly scalable, distributed graph database that integrates with various backends, including HBase and Cassandra. HugeGraph [9] is an open-source, distributed graph database designed to handle large-scale graphs efficiently. TinkerGraph [60] is a lightweight, embedded graph database that serves as a reference implementation for the TinkerPop graph processing framework.
- Testing Environment.** All experiments are conducted on a server with 96 physical cores running at 2.60 GHz. The machine runs Ubuntu 22.04 and is equipped with 256 GB of RAM.
- Baseline Approaches.** We select seven publicly available testing approaches targeting Cypher- and Gremlin-compatible GDBMSs as baselines: GDSmith [21], GDBMeter [29], Gamera [73], GraphGenie [25], GRev [42], GraspDB [39], and Grand [72]. Among these approaches, GDSmith and Grand are differential testing approaches, targeting Cypher and Gremlin, respectively. The remaining approaches support Cypher and all employ metamorphic testing strategies to address the oracle problem, while Gamera additionally supports Gremlin.

4.1 RQ1: Real-world Applicability

Evaluation methodology. To evaluate the real-world applicability of PATHTest, we apply it to seven popular real-world GDBMSs. As a metamorphic testing approach, PATHTest is capable of detecting both unexpected errors and logical bugs. Each of them is later checked, reproduced, and reported to the developers. To improve

Table 2: Previously Unknown Bug Detected by PATHTest

GDBMS	Detected	Bug status		Bug type	
		Confirmed	Fixed	Logic	UE
Neo4j [55]	6	6	6	1	5
MemGraph [45]	14	9	0	11	3
AgensGraph [65]	10	5	3	5	5
NebulaGraph [22]	5	5	0	3	2
JanusGraph [10]	1	0	0	1	0
HugeGraph [9]	4	4	2	2	2
TinkerGraph [60]	1	0	0	1	0
Total	41	29	11	24	17

UE: Unexpected Error (e.g., runtime exception, crash, or internal error).

the quality of our bug reports, a systematic process is applied: ① We manually minimize each bug report to improve reproducibility and facilitate bug fixing. ② We further manually perform root cause analysis to filter out test cases triggered by the same issue. ③ We then cross-check public issues to prevent duplicate submissions, and submit new unique bugs with detailed reports and reproduction steps. ④ Among the reported bugs, only those explicitly acknowledged by the developers are marked as confirmed or fixed.

Results. Table 2 summarizes the number and status of unknown bugs detected by PATHTest across the seven evaluated GDBMSs. In total, PATHTest uncovers 46 unique bugs, among which 41 are previously unknown. The remaining five bugs had already been reported or immediately fixed before the detection of PATHTest, and are therefore excluded from our statistics. By now, among the 41 newly identified bugs, 24 are logic bugs, and 17 are unexpected errors. To better understand their root causes, we conducted a further manual analysis based on error messages, query plans, execution traces, and developer feedback.

Logic bugs. Among the 24 logic bugs, 10 are triggered by fixed-length queries and 14 by variable-length queries. For fixed-length pattern-related bugs, 3 are caused by index optimization logic errors, and 4 by subquery usage, where the outer query failed to reference results produced by the subquery, resulting in empty intermediate results in the outer query. The remaining 3 are caused by incorrect query plans that failed to preserve the intended semantics. For variable-length pattern-related bugs, 8 are due to invalid path matches resulting from incorrect query plans, while 6 are due to inaccurate execution-time optimizations.

Unexpected errors. As for the 17 unexpected errors, 5 are caused by planning failures where the query engine failed to generate a valid plan. Another 3 stem from subqueries, causing inaccessible variables or transaction conflicts, and three result from an optimization failure. The root causes of the remaining 9 errors remain unclear due to limited developer feedback and insufficient diagnostic information in error messages.

These diverse causes of query correctness issues, ranging from semantic mismatches to planning and execution-time anomalies, highlight the breadth of query-processing scenarios of PATHTest.

Key Findings. Based on the above analysis, we summarize three key findings: ① the 11 logic bugs and five unexpected errors caused by incorrect query plans show that our transformation strategies

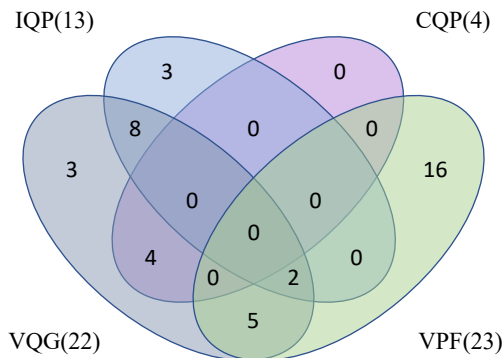


Figure 3: Venn diagram of bug-related transformation.

generate structurally diverse and semantically rich queries, which in turn stress the query planner and reveal latent flaws. ② the six subquery-related bugs underscore the strength of IQP in revealing defects tied to subqueries, an essential but largely overlooked class of queries in prior testing approaches [21, 25, 29, 39, 42, 73]. ③ the eight optimization-related bugs found in variable-length queries indicate that variable-length queries continue to be a considerable source of correctness problems due to the inherent complexity of their execution and performance-oriented optimizations. These optimizations, while critical for scaling to large graphs, inevitably introduce non-trivial complications and significantly challenge the correctness assurance of these queries. Furthermore, our system-specific extensions to the VPF strategy also demonstrated compelling effectiveness in uncovering database-specific bugs. For example, in Neo4j, the use of ALL_SHORTEST in folded path queries led to the detection of two additional bugs: one resulting in execution failure and the other in query planning failure. These findings further demonstrate the real-world applicability of PATHTest in detecting practical bugs in modern GDBMSs.

4.2 RQ2: Ablation Evaluation

Evaluation methodology. The variable-length query generation component VQG, together with the three transformation rules (VPF, IQP, and CQP), are the four novel components proposed by PATH-Test and collectively contribute to the overall effectiveness. To further understand the individual and combined contributions, we conduct an ablation evaluation of the first-found bugs reported in Table 2. Specifically, we minimize the triggering queries to identify the component directly related to each detected bug. A component is deemed directly related to a bug if removing the mutation (by VQG) or transformation (by VPF, IQP, or CQP) from the triggering query prevents the bug from occurring.

Bug-related analysis. Figure 3 presents a Venn diagram illustrating the distribution of independent and combined contributing bugs across the four components. As presented, 23 unique bugs could be detected solely through VPF, accounting for more than half of all bugs detected by PATHTest. Notably, 5 out of the 23 bugs could only be exposed after applying heterogeneous-label generalization. These bugs cannot be triggered by directly mutating fixed-length queries, since they emerge only when intermediate nodes and edges

Table 3: Results on Comparison with Cypher-supporting Baselines

Approach	Neo4j			Memgraph			AgensGraph			NebulaGraph		
	Total	Unique	False Alarm	Total	Unique	False Alarm	Total	Unique	False Alarm	Total	Unique	False Alarm
PATHTest	305	5	0	359	8	0	577	6	0	590	5	0
GDSmith [21]	1000	0	1000	1000	0	1000	-	-	-	-	-	-
GDBMeter [29]	215	2	206	-	-	-	-	-	-	-	-	-
GRev [42]	0	0	0	12	1	0	-	-	-	150	3	0
GraphGenie [25]	1	0	0	-	-	-	397	0	397	-	-	-
Gamera [73]	114	1	111	-	-	-	-	-	-	-	-	-
GraspDB [39]	56	3	7	1000 [†]	0	1000 [†]	1000 [†]	0	1000 [†]	-	-	-

Table 4: Results on Comparison with Gremlin-based Baselines

Approach	JanusGraph			HugeGraph			TinkerGraph		
	Total	Unique	False Alarm	Total	Unique	False Alarm	Total	Unique	False Alarm
PATHTest	8	1	0	220	8	0	4	1	0
Grand [72]	1000	0	155	1000	4	155	1000	0	155
Gamera [73]	347	0	347	1000 [†]	0	1000 [†]	330	0	330

are folded into variable-length paths while preserving heterogeneous label constraints via predicates. These results highlight the effectiveness of PATHTest in revealing bugs specific to complex variable-length path handling. In addition, VQG and IQP each independently detect 3 unique bugs. For VQG, 2 bugs are triggered as variables bound in variable-length patterns are not properly carried over by the WITH clause to subsequent clauses, which is an issue that cannot arise with fixed-length patterns. The remaining bug is an unexpected error triggered by variable-length patterns. For IQP, 3 unique bugs are unexpected errors related to subqueries. The remaining 10 IQP-related bugs, as well as all 4 bugs from CQP, could only be detected when VQG is involved. This is because both IQP and CQP identify logic bugs by referring to the original generated variable-length patterns by VQG as test oracles. In addition to the bugs discussed above, 5 additional bugs arise from the combined effects of VQG and VPF. While both VQG and VPF generate variable-length queries, they capture complementary aspects of query semantics and jointly contribute to the detection of these bugs. Impressively, 2 unique logic bugs can only be detected by combining VQG, VPF, and IQP: VQG first generates a variable-length pattern, which is then partitioned into a subquery by IQP; VPF afterwards folds patterns within the subquery into a variable-length form, eventually triggering these 2 subquery-related bugs.

The above findings demonstrate both the independent and combined effectiveness of the four novel components, which contribute together to the superior overall effectiveness of PATHTest.

4.3 RQ3: Comparison Evaluation

Evaluation methodology. We conduct an empirical comparison between PATHTest and baselines to highlight their practical differences, with our best effort to ensure fairness. On the one hand, we execute PATHTest and all baseline approaches under the same testing environments. For Cypher-supporting GDBMSs, we conduct a 24-hour test, which is commonly adopted in prior studies [27, 33, 69].

For Gremlin-based GDBMSs, we conduct 10 rounds, each generating 1,000 queries, which is a commonly used experimental setup for Gremlin [72, 73]. On the other hand, we investigate the earliest versions of the GDBMSs where the 24 logic bugs reported by PATHTest could be triggered, analyzing whether these bugs were introduced before the baselines published. This retrospective analysis, frequently employed in previous studies [27, 69], examines whether PATHTest revealed previously overlooked logic bugs.

Detected unique bugs. Table 3 and Table 4 present the experimental comparison results, summarizing the total number of bug reports analyzed, the number of unique bugs, and false positives identified through our manual analysis. The manual analysis followed a standardized report validation and deduplication process. Three Ph.D. researchers independently analyze each bug report by executing it to identify the error type, simplify the test case, and verify whether it conforms to the defined metamorphic relation. Relevant details, including error messages, query components, and mutation operators, are recorded to identify duplicates and guide subsequent deduplication. Only when all three researchers reach a consensus is the report considered a new true bug. During the 24-hour evaluation for Cypher-supporting GDBMSs, GDSmith, GDBMeter, GRev, GraphGenie, Gamera, GraspDB, and PATHTest produce 2,489; 215; 162; 398; 114; 10,056; and 1,831 bug reports, respectively. Meanwhile, in the ten-round evaluation for Gremlin-based GDBMSs, Grand, Gamera, and PATHTest produce 1,205; 10,677; and 232 bug reports, respectively. For results with an excessively large number of reports, it is impractical to analyze all manually. As a result, for any system that reports over 1,000 bugs on a single GDBMS, we randomly select 1,000 for manual inspection. This approach ensures both the feasibility of the analysis and the reliability of the evaluation results. The large number of reports from GraspDB and Gamera are mainly due to their outdated implementations for these GDBMSs and are no longer compatible with the latest versions, resulting in a large volume of syntax and semantic errors. Therefore, we include its results here for completeness but do not further analyze them.

Among all the reports produced by PATHTest under comparison settings, we identified 5, 8, 6, 5, 1, 8, and 1 unique bugs on Neo4j, MemGraph, AgensGraph, NebulaGraph, JanusGraph, HugeGraph, and TinkerGraph, respectively, yielding 34 unique bugs in total, among which 18 are logic bugs. Our detailed investigation of the logic bugs reveals that those related to fixed-length queries can *only* be exposed when paired with result-equivalent variable-length queries serving as test oracles, and vice versa. Therefore, without cross-reference testing oracles between fixed-length and variable-length queries, all 18 logic bugs cannot be revealed by all existing metamorphic testing approaches, demonstrating the unique ability of PATHTest. We also tried differential testing and further executed the identified bug-triggering across different GDBMSs. However, many of them could not be executed uniformly, as some systems lack full support for subqueries while others implement different characteristics for variable-length queries. These inconsistencies hinder the same query from being executed comparably, resulting in limited overlap between query results and thus restricting the effectiveness of differential testing in revealing such logic bugs. Consequently, only 4 logic bugs detected by PATHTest could be reproduced via differential testing.

Integrating all the baseline approaches, we identified 2, 4, 1, 3, and 4 unique bugs in GDBMeter, GRev, Gamera, GraspDB, and Grand, respectively. For GDBMeter, 2 unique bugs were found on Neo4j: a logic bug and an unexpected error. The logic bug is detected through its partitioning-based oracle, which decomposes a query Q into three subqueries (Q_P , $Q_{\neg P}$, and Q_{null}) corresponding to the possible outcomes of predicate P : TRUE, FALSE, and NULL. Since our metamorphic rules do not involve predicate-level transformations, such bugs fall outside the detection scope of our approach and thus cannot be exposed by PATHTest. For GRev, one logic bug was detected on MemGraph and three on NebulaGraph. The transformation rules in GRev rely on equivalently restructuring the pattern, such as converting a pattern $(a) - [] \rightarrow (b)$ into $(a), (b) < - [] - ()$, (a) , thereby altering the query plan. Our approach, in contrast, transforms the bug query patterns into result-equivalent variable-length patterns, which enabled us to successfully reproduce all four of these logic bugs. As for GraspDB, two logic bugs and one unexpected error were detected on Neo4j. Since these logic bugs are triggered by write-related metamorphic rules, which are orthogonal to the focus of our approach, PATHTest was unable to reproduce them currently. For Grand, 4 unique unexpected errors were detected, which were also revealed by PATHTest. For all six unexpected errors mentioned above and one identified by Gamera, as well as the 16 revealed by PATHTest, their primary issue lies in query generation, which is independent of the test oracles. Additionally, both GDSmith and GraphGenie failed to detect any bugs during the evaluation. Our further investigation reveals that most of the bugs reported by GDSmith and GraphGenie in their papers have already been fixed; it is reasonable that hardly any new bugs can be detected based on their oracles in the latest versions. Furthermore, since GDSmith is a differential-testing-based approach requiring comparisons across multiple GDBMSs, its effectiveness in our benchmark is limited by the fact that it only supports two GDBMSs in our benchmark.

In total, PATHTest discovered 34 unique bugs across all seven GDBMSs under comparison settings. In comparison, GDBMeter, GRev, Gamera, GraspDB, and Grand identified only 2, 4, 1, 3, and

4 unique bugs, respectively, while the other baselines detected none. Notably, none of these approaches detected any genuine bugs related to variable-length queries, whereas PATHTest identified 16 such cases. Even within the fixed-length query space, while four were also detected by Grand, PATHTest exclusively uncovered another 14 previously unknown bugs. Such findings demonstrate the unique ability of PATHTest to expose correctness issues overlooked by existing approaches, across both fixed-length and variable-length queries, revealing both semantic inconsistencies and execution anomalies across GDBMSs.

False Alarms. Among baseline approaches, GDSmith and Grand reported a large number of false alarms. For GDSmith, these false positives mainly stem from subtle semantic differences in Cypher between Neo4j and MemGraph, a well-known limitation of differential testing for GDBMSs [25, 39, 73]. Listing 1 shows a representative case where different numeric return types lead to a spurious discrepancy. For Grand, most false alarms are caused by syntactically invalid queries and its coarse-grained comparison strategy [14], which directly compares execution outcomes as raw strings without distinguishing error types across GDBMSs, thereby incorrectly reporting syntax errors as bugs. Although our PATHTest is based on GDSmith for Cypher and Grand for Gremlin, it employs metamorphic testing, which avoids false alarms caused by discrepancies among different GDBMSs. Moreover, due to the rigorously designed variable-length query generation and result-equivalent transformation rules, PATHTest did not produce any false alarms. Consequently, all issues reported by PATHTest correspond to real bugs in the target GDBMSs (including duplicates).

```
1 WITH avg(913440431) AS a0 RETURN DISTINCT a0
2 -- {a0:913440431.0} on \database{Neo4j},
3 -- {a0:913440431} on \database{MemGraph}
```

Listing 1: False alarm in GDSmith

```
1 CREATE (:z {Hae:(4865233778139072963)% (size(""))})
2 -- ArithmeticError: / by zero
```

Listing 2: False alarm in GDBMeter and Gamera

As for the other baselines, GDBMeter’s transformation rules are designed to preserve equivalence and thus are not expected to produce false positives in theory. However, our analysis reveals that most of its reports are actually false alarms, mainly caused by missing constraints on divisor values during query generation and result in division-by-zero errors, as shown in Listing 2. Since Gamera employs GDBMeter as its Cypher query generator, it exhibits the same issue in Cypher. In addition, in the Gremlin experiments on JanusGraph and TinkerGraph, all reports from Gamera are time-out exceptions that are also categorized as false alarms. All reports generated by GraphGenie are false positives, caused by two main issues [15]. First, it ignores semantic discrepancies between MATCH and OPTIONAL MATCH. Second, an incorrect assumption that count() and count(input) are equivalent, whereas count() includes rows returning null and count(input) ignores them (count(null) returns 0). GraspDB’s false alarms stem from its mutation rule that involves creating a node and then deleting it, which introduces new variables into the query and alters its original semantics [39]. In contrast, GRev does not produce any false positives in our evaluation.

Table 5: Latency of the bug found by PATHTest

GDBMS	Found	Bug-involved year		Longest latency
		<2024	<2023	
Neo4j [55]	1	1	1	6 years
Memgraph [45]	11	11	5	2 years
AgensGraph [65]	5	4	4	3 years
NebulaGraph [22]	3	3	3	3 years
JanusGraph [10]	1	1	1	3 years
HugeGraph [9]	2	2	2	4 years
TinkerGraph [60]	1	1	1	3 years

Introduced Times of Bugs. We categorize the baselines by publication year, with those from 2023 (GDSmith, GDBMeter, Gamera) and 2024 (GraphGenie, GRev, GraspDB), and analyzed when the logic bugs detected by PATHTest were introduced. A summary of these findings is presented in Table 5. Among the 24 logic bugs, 23 can be reproduced on GDBMS versions released before 2024, and 17 existed even before 2023. For the 2023 baselines, GDSmith supports Neo4j and MemGraph, and 6 out of 12 bugs found on these systems were introduced before its publication. Both GDBMeter and Gamera support Neo4j, and the single bug found on this system predates both approaches. For the 2024 baselines, GraphGenie supports Neo4j and AgensGraph, with 5 out of 6 bugs introduced before its publication. GRev supports Neo4j, MemGraph, and NebulaGraph, and all 15 bugs identified on these systems predate its release. GraspDB supports Neo4j, MemGraph, and AgensGraph, with 16 out of 17 bugs already present before its publication. In addition, Gamera also supports Gremlin-based GDBMSs, and we found four logic bugs on JanusGraph, HugeGraph, and TinkerGraph; all of these bugs were introduced before the publication of Gamera. Moreover, Grand, a differential testing approach for Gremlin published in 2022, supports the same three systems. By manually inspecting the affected versions, we confirmed that all four Gremlin bugs identified by PATHTest predate the release of Grand. Notably, the longest-hidden logic bug was found in Neo4j and had been deeply hidden for over six years. These findings demonstrate that PATHTest can uncover numerous long-latent logic bugs overlooked by existing approaches, highlighting its effectiveness and uniqueness.

4.4 Case Study of Real-World Bugs

In this section, we present representative and interesting categories of bugs identified by PATHTest. For clarity, query examples are simplified by omitting graph creation statements.

```

1 OPTIONAL MATCH (n3)-[]->(n3:L5)<-[]-(n5) RETURN *;
2 // empty
3 OPTIONAL MATCH (n3)-[m0*1..1]->(n3:L5)<-[m1*1..2]-(n5) WHERE
  length(m0)=1 AND length(m1)=1 WITH n3, n5 RETURN *;
4 // ERROR: column "n3" does not exist

```

Listing 3: Unexpected errors detected by VPF

A case from VPF. Listing 3 illustrates a bug triggered by a query involving variable-length patterns [13]. During the VPF phase, our system transforms the original query (line 1) into a result-equivalent form (line 3) that explicitly constrains the path length via two quantified path expressions. Although the rewritten query is semantically valid, AgensGraph fails to execute it, reporting that variable n3

does not exist, even though it is clearly bound in the MATCH clause. Upon investigation, we traced and located the root cause to AgensGraph’s internal handling of Variable-Length Expressions (VLEs) within join operations. Specifically, the planner fails to correctly propagate bound node variables when they appear on both ends of a VLE, particularly in self-joins with label constraints. This issue arises as the internal representation of join targets becomes inaccessible during join list construction, which prevents proper binding resolution. While the patch avoids execution failure by bypassing the invalid join construction, it does not resolve the underlying issue: the planner’s inability to correctly bind node variables when variable-length edges are unnamed and involved in complex join structures, such as self-referential patterns with label filters. This case highlights how variable-length pattern rewrites, especially those involving self-referential or filtered paths, can expose deep flaws in the query planner’s variable scoping mechanism.

```

1 g.E().outV().repeat(__.both()).times(1).where(__.outE().
  count().is(lt(-3))).count() // Result: 305
2 g.E().outV().match(__.as('start').both().where(__.outE().
  count().is(lt(-3))).as('end')).select('end').count()
  // Result: 0

```

Listing 4: Logic bug detected by IQP

A case from IQP. Listing 4 shows a logic bug uncovered by IQP on HugeGraph [9], a Gremlin-based GDBMS. The original query performs a one-hop neighborhood expansion using a variable-length traversal `repeat(__.both()).times(1)`, followed by a filtering condition. The rewritten query applies IQP by replacing the variable-length traversal with a result-equivalent fixed-length traversal, which is realized through an explicit `both` step combined with an inner traversal defined via `match`. Although the two queries should return the same result, the original query returns 305, whereas the rewritten query yields zero. Further analysis reveals that the bug is rooted in an index-related optimization issue in the execution of the original query. Specifically, HugeGraph pushes the filtering predicate in the original traversal into the backend and enforces index-based evaluation, which leads to incorrect pruning of intermediate results. In contrast, the rewritten query embeds the same predicate inside the inner traversal of `match`, where it is treated as an in-memory filter evaluated after data retrieval. As a result, the planner bypasses index enforcement for the predicate, avoiding the faulty index optimization and yielding the right results. The issue has been reported to HugeGraph [16].

```

1 MATCH (n0:L6)-[*1..3]->(n1)RETURN 1 AS a4
2 // Result: 0 rows
3 MATCH (n0:L6)-[]->(n1)RETURN 1 AS a4
4 UNION ALL
5 MATCH (n0:L6)-[]->()-[]->(n1)RETURN 1 AS a4
6 UNION ALL
7 MATCH (n0:L6)-[]->()-[]->()-[]->(n1)RETURN 1 AS a4
8 // Result: 9 rows

```

Listing 5: Logic bug detected by CQP

A case from CQP. Listing 5 demonstrates a logic inconsistency in AgensGraph triggered by CQP [12]. The original query uses a variable-length pattern `(n0:L6)-[*1..3]->(n1)` to find paths up to 3 hops from nodes labeled L6, but it returns no results. However,

after rewriting it as a result-equivalent disjunction of three fixed-length queries using UNION ALL (lines 3–7), it returns 9 rows, revealing a discrepancy between fixed and variable-length pattern evaluations. Our investigation found the root cause lies in AgensGraph’s internal handling of variable-length queries during query planning. For fixed-length queries such as $(n0)-[]->()-[]->(n1)$ is processed as a series of joins over graph edges. Conversely, variable-length queries like $(n0)-[*1..3]->(n1)$ are treated as joins over paths. As noted in the relevant fix commit [11], unless an explicit path variable is declared, the planner does not treat anonymous variable-length patterns as path objects. Consequently, the join operates on an unresolved structure, silently failing to produce matches. This causes a logic bug where the variable-length query returns no results, while its fixed-length rewritten version returns correct matches. In short, the issue stems from the optimizer treating variable-length queries differently from fixed-length queries.

5 DISCUSSION ON GQL STANDARD

With the recent publication of the ISO/IEC 39075:2024 standard for graph query language, GQL [24], it is important to highlight the compatibility of our proposal with this emerging standard. GQL adopts a pattern-matching syntax similar to Cypher [23], where pattern matching is used for fixed-length queries to define exact structural patterns. Additionally, GQL introduces a regex-like syntax for variable-length queries to enhance flexibility in path matching. For example, the syntax $-[e]->\{1,3\}$ specifies path lengths from 1 to 3, $-[e]->+$ supports paths of at least one hop, while $-[e]->*$ supports paths of arbitrary length. Although these enhancements expand the expressiveness of the query language, such syntactical differences do not affect the applicability of our approach. The core transformations we present for fixed-length and variable-length queries can be adapted to GQL with slight engineering efforts. Consequently, our approach can be seamlessly integrated into future graph databases adopting this standard.

6 RELATED WORK

Differential Testing. Differential testing [44] is widely adopted for detecting bugs by executing the same input across multiple systems and identifying discrepancies in their outputs [28, 53]. For GDBMSs, Grand [72] introduces a model-based query generation approach to produce valid Gremlin queries. RD2 [67] applies randomized differential testing to GDBMSs that support SPARQL [63]. GD-Smith [21] targets Cypher, introducing a query generation framework that first constructs query skeletons and then populates them with graph-guided patterns and data-guided predicates, achieving sophisticated non-empty-result queries. However, differential testing is inherently constrained to the common feature set shared across systems, making it limited for system-specific behaviors. Furthermore, differential testing cannot reveal bugs consistently present across all tested systems.

Metamorphic Testing. Metamorphic testing [5] addresses the oracle problem [4] by verifying whether outputs satisfy predefined metamorphic relations under systematic input transformations, which has demonstrated compelling success in both RDBMSs [2, 18, 26, 27, 38, 51, 52, 54, 69] and GDBMSs [25, 29, 39, 70, 71, 73].

For GDBMSs, most metamorphic testing approaches mainly operate on expression- or clause-level. GDBMeter [29] adopts the core idea of TLP [52] to detect predicate-related bugs by transforming the original query into a union of three separate queries with three non-intersecting logic branches, namely p , NOT p , and p IS NULL. Gamera [73] exploits native graph structural properties (e.g., reachability, parent-child, neighbor relations) to design three classes of graph-aware metamorphic relations for detecting logic bugs. GraspDB [39] defines three classes of mutation rules over writing clauses to detect writing-related bugs. Dinkel [66] optimizes existing query generation methods and defines a series of result-equivalent metamorphic rules at both the clause and expression levels, enabling it to cover nearly all query scopes and detect a great number of bugs. Qudi [70] targets Gremlin, which disassembles queries into atomic traversals while maintaining semantics to detect logic bugs. In contrast, GraphGenie [25] and Grev [42] can explicitly manipulate graph patterns for fixed-length queries. GraphGenie defines a set of mutation rules for graph patterns and detects logic bugs by checking whether the resulting outputs satisfy expected relations (i.e., equivalence, subset, or superset). GRev likewise operates on graph patterns; it reconstructs Cypher patterns via an Abstract Syntax Graph and generates a large set of equivalent pattern variants to detect logic bugs.

Although existing metamorphic testing approaches effectively diversify query execution plans of fixed-length queries through various mutation rules, they remain unable to modify the semantics of variable-length queries, thus incapable of testing behaviors related to such semantics. PATHTest addresses such limitations by bridging fixed-length and variable-length queries via result-equivalent transformations, not only enabling effective testing of variable-length queries, but also extending the testing scope of fixed-length queries with more abundant test oracles. Consequently, PATHTest uniquely detects numerous bugs missed by existing approaches, broadening the testing boundaries for modern GDBMSs.

7 CONCLUSION

We propose PATHTest, a metamorphic testing approach targeting two core features of graph query languages, graph pattern matching and graph navigation, which serve as the underlying mechanisms for fixed-length and variable-length queries, respectively. With the proposed novel *folding* and *partition* techniques, PATHTest generates result-equivalent variants, enabling effective metamorphic testing. Evaluation conducted on seven popular production-grade GDBMSs demonstrates the effectiveness of PATHTest in real-world practice, with 41 previously unknown bugs revealed. Additionally, the comparison with seven state-of-the-art testing approaches further demonstrates the effectiveness and uniqueness of PATHTest in uncovering missed bugs by existing approaches while maintaining a zero false-positive rate.

ACKNOWLEDGMENTS

This research is supported by the National Natural Science Foundation of China (62372227 and 62402213) and the “111 Center” (No. B26023).

REFERENCES

- [1] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoč. 2016. Foundations of Modern Query Languages for Graph Databases. *ACM Computing Surveys (CSUR)* 50 (2016), 1–40. <https://api.semanticscholar.org/CorpusID:13526884>
- [2] Jinsheng Ba and Manuel Rigger. 2024. CERT: Finding Performance Issues in Database Systems Through the Lens of Cardinality Estimation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 133, 13 pages. <https://doi.org/10.1145/3597503.3639076>
- [3] Pablo Barceló. 2013. Querying graph databases. *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems* (2013). <https://api.semanticscholar.org/CorpusID:208007633>
- [4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. <https://doi.org/10.1109/TSE.2014.2372785>
- [5] Tsong Yueh Chen, Shing Chi Cheung, and Siu ming Yiu. 2020. Metamorphic Testing: A New Approach for Generating Next Test Cases. *ArXiv abs/2002.12543* (2020). <https://api.semanticscholar.org/CorpusID:15467386>
- [6] Xiaojun Chen, Shengbin Jia, and Yang Xiang. 2020. A review: Knowledge reasoning over knowledge graph. *Expert systems with applications* 141 (2020), 112948.
- [7] Xin Chen, You Peng, Sibow Wang, and Jeffrey Xu Yu. 2022. DLCR: Efficient Indexing for Label-Constrained Reachability Queries on Large Dynamic Graphs. *Proc. VLDB Endow.* 15 (2022), 1645–1657. <https://api.semanticscholar.org/CorpusID:250095289>
- [8] Cloc. 2025. Count Lines of Code. <https://github.com/AIDaniel/cloc>.
- [9] The Apache Software Foundation. 2025. HugeGraph. <https://hugegraph.apache.org/>.
- [10] The Linux Foundation. 2025. JanusGraph. <https://janusgraph.org/>.
- [11] GitHub. 2025. AgensGraph issues #610. <https://github.com/skaiworldwide-oss/agensgraph/pull/610>.
- [12] GitHub. 2025. AgensGraph Issues #728. <https://github.com/skaiworldwide-oss/agensgraph/issues/728>.
- [13] GitHub. 2025. AgensGraph Issues #731. <https://github.com/skaiworldwide-oss/agensgraph/issues/731>.
- [14] GitHub. 2026. Grand issues #1. <https://github.com/choeoe/Grand/issues/1#issuecomment-1305004036>.
- [15] GitHub. 2026. GraphGenie issues #3. <https://github.com/YuanchengJiang/GraphGenie/issues/3#issuecomment-1805133154>.
- [16] GitHub. 2026. HugeGraph Issues #2930. <https://github.com/apache/incubator-hugegraph/issues/2930>.
- [17] The PostgreSQL Global Development Group. 2026. PostGreSQL. <https://www.postgresql.org/>.
- [18] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 345–358. <https://www.usenix.org/conference/atc23/presentation/hao>
- [19] Richard Henderson. 2020. Using graph databases to detect financial fraud. *Computer Fraud & Security* 2020, 7 (2020), 6–10. [https://doi.org/10.1016/S1361-3723\(20\)30073-7](https://doi.org/10.1016/S1361-3723(20)30073-7)
- [20] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, et al. 2021. Knowledge graphs. *ACM Computing Surveys (Csur)* 54, 4 (2021), 1–37.
- [21] 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* (Seattle, WA, USA) (ISSTA 2023). Association for Computing Machinery, New York, NY, USA, 163–174. <https://doi.org/10.1145/3597926.3598046>
- [22] Vesoft Inc. 2025. NebulaGraph. <https://www.nebula-graph.io/>.
- [23] ISO/IEC. 2024. ISO/IEC 39075 Database Language GQL. <https://jtc1info.org/document/gql-database-language/> Accessed: 2026-01-22.
- [24] ISO/IEC. 2024. ISO/IEC 39075:2024 Information technology — Database languages — GQL. <https://www.iso.org/standard/76120.html> Accessed: 2026-01-22.
- [25] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland H. C. Yap, Zhenkai Liang, and Manuel Rigger. 2024. Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 46, 12 pages. <https://doi.org/10.1145/3597503.3623307>
- [26] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 397–417. <https://www.usenix.org/conference/osdi23/presentation/jiang>
- [27] Zu-Ming Jiang and Zhendong Su. 2024. Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 821–835. <https://www.usenix.org/conference/osdi24/presentation/jiang>
- [28] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. APOLLO: automatic detection and diagnosis of performance regressions in database systems. *Proc. VLDB Endow.* 13, 1 (Sept. 2019), 57–70. <https://doi.org/10.14778/3357377.3357382>
- [29] Matteo Kamm, Manuel Rigger, Chengyu Zhang, 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* (Seattle, WA, USA) (ISSTA 2023). Association for Computing Machinery, New York, NY, USA, 140–149. <https://doi.org/10.1145/3597926.3598044>
- [30] Arijit Khan, Nan Li, Xifeng Yan, Ziyu Guan, Supriyo Chakraborty, and Shu Tao. 2011. Neighborhood based fast graph search in large networks. In *ACM SIGMOD Conference*. <https://api.semanticscholar.org/CorpusID:7385724>
- [31] Arijit Khan, Yinghui Wu, Charu C. Aggarwal, and Xifeng Yan. 2013. NeMa: Fast Graph Search with Label Similarity. *Proc. VLDB Endow.* 6 (2013), 181–192. <https://api.semanticscholar.org/CorpusID:10828525>
- [32] Stephen Cole Kleene. 1956. *Representation of events in nerve nets and finite automata*. Vol. 34. Princeton University Press Princeton.
- [33] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) (CCS '18). Association for Computing Machinery, New York, NY, USA, 2123–2138. <https://doi.org/10.1145/3243734.3243804>
- [34] Hyeyoung Ko, Suyeon Lee, Yoonseo Park, and Anna Choi. 2022. A survey of recommendation systems: recommendation models, techniques, and application fields. *Electronics* 11, 1 (2022), 141.
- [35] Dexter Kozen. 1990. On Kleene Algebras and Closed Semirings. In *International Symposium on Mathematical Foundations of Computer Science*. <https://api.semanticscholar.org/CorpusID:2072425>
- [36] Dexter Kozen. 1994. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and computation* 110, 2 (1994), 366–390.
- [37] Mingdao Li, Peng Peng, Zheyuan Hu, Lei Zou, and Zheng Qin. 2024. Variable-Length Path Query Evaluation Based on Worst-Case Optimal Joins. *2024 IEEE 40th International Conference on Data Engineering (ICDE)* (2024), 3311–3323. <https://api.semanticscholar.org/CorpusID:271405900>
- [38] Jie Liang, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Chengnian Sun, and Yu Jiang. 2024. Mozi: Discovering DBMS Bugs via Configuration-Based Equivalent Transformation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 135, 12 pages. <https://doi.org/10.1145/3597503.3639112>
- [39] 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* (Vienna, Austria) (ISSTA 2024). Association for Computing Machinery, New York, NY, USA, 666–677. <https://doi.org/10.1145/3650212.3680311>
- [40] Red Gate Software Ltd. 2025. DB-Engines Ranking of Graph DBMS. <https://db-engines.com/en/ranking/graph+dbms>.
- [41] Bingqing Lyu, Xiaoli Zhou, Longbin Lai, Yufan Yang, Yunkai Lou, and Yongfei Liu. 2024. Enhancing Neo4j Query Efficiency with Seamless Integration of the GOpt Optimization Framework. In *VLDB Workshops*. <https://api.semanticscholar.org/CorpusID:273878653>
- [42] Qiuyang Mang, Aoyang Fang, Boxi Yu, Hanfei Chen, and Pinjia He. 2024. Testing Graph Database Systems via Equivalent Query Rewriting. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 143, 12 pages. <https://doi.org/10.1145/3597503.3639200>
- [43] MarketsandMarkets. 2025. Graph Database Market. <https://www.marketsandmarkets.com/Market-Reports/graph-database-market-126230231.html>.
- [44] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107. <http://dblp.uni-trier.de/db/journals/dtj/dtj10.html#McKeeman98>
- [45] Memgraph. 2025. MemGraph. <https://memgraph.com/>.
- [46] Alberto O. Mendelzon and Peter T. Wood. 1989. Finding Regular Simple Paths in Graph Databases. In *VLDB*. Morgan Kaufmann, 185–193.
- [47] Justin J. Miller. 2013. Graph Database Applications and Concepts with Neo4j. <https://api.semanticscholar.org/CorpusID:17523312>
- [48] Neo4j. 2025. OpenCypher. <https://opencypher.org/>.
- [49] You Peng, Xuemin Lin, Ying Zhang, Wenjie Zhang, and Lu Qin. 2021. Answering reachability and K-reach queries on large graphs with label constraints. *The VLDB Journal* 31 (2021), 101–127. <https://api.semanticscholar.org/CorpusID:239664533>

- [50] You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and W. Zhang. 2020. Answering billion-scale label-constrained reachability queries within microsecond. *Proceedings of the VLDB Endowment* 13 (2020), 812–825. <https://api.semanticscholar.org/CorpusID:211267332>
- [51] 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* (Virtual Event, USA) (ESEC/FSE 2020). Association for Computing Machinery, New York, NY, USA, 1140–1152. <https://doi.org/10.1145/3368089.3409710>
- [52] 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>
- [53] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 618–622.
- [54] Jiansen Song, Wensheng Dou, Yu Gao, Ziyu Cui, Yingying Zheng, Dong Wang, Wei Wang, Jun Wei, and Tao Huang. 2024. Detecting Metadata-Related Logic Bugs in Database Systems via Raw Database Construction. *Proc. VLDB Endow.* 17, 8 (April 2024), 1884–1897. <https://doi.org/10.14778/3659437.3659445>
- [55] The Neo4j Team. 2025. Neo4j. <https://neo4j.com/>.
- [56] The Neo4j Team. 2025. The Neo4j Manual v5. <https://neo4j.com/docs/cypher/>.
- [57] TigerGraph. 2025. GSQL: Graph Query Language. <https://www.tigergraph.com/gsql/>.
- [58] TigerGraph. 2025. TigerGraph. <https://www.tigergraph.com/>.
- [59] Apache TinkerPop. 2025. Gremlin Query Language. <https://tinkerpop.apache.org/gremlin.html>.
- [60] Apache TinkerPop. 2025. TinkerGraph. <https://tinkerpop.apache.org/>.
- [61] Vesoft. 2025. Nebula Graph Query Language. <https://docs.nebula-graph.io/2.6.1/3.ngql-guide/1.ngql-overview/1.overview/>.
- [62] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. 2010. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th Annual ACM Southeast Conference* (Oxford, Mississippi) (ACMSE '10). Association for Computing Machinery, New York, NY, USA, Article 42, 6 pages. <https://doi.org/10.1145/1900008.1900067>
- [63] W3C. 2013. SPARQL Query Language for RDF. <https://www.w3.org/TR/rdf-sparql-query/>.
- [64] Peter T. Wood. 2012. Query languages for graph databases. *SIGMOD Rec.* 41 (2012), 50–60. <https://api.semanticscholar.org/CorpusID:267795861>
- [65] SKAI Worldwide. 2025. AgensGraph. <https://github.com/skaiworldwide-oss/agensgraph>.
- [66] Celine Wüst, Zu-Ming Jiang, and Zhendong Su. 2025. Dinkel: State-Aware and Granular Framework for Validating Graph Databases. arXiv:2408.07525 [cs.DB] <https://arxiv.org/abs/2408.07525>
- [67] Rui Yang, Yingying Zheng, Lei Tang, Wensheng Dou, Wei Wang, and Jun Wei. 2023. Randomized Differential Testing of RDF Stores. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 136–140. <https://doi.org/10.1109/ICSE-Companion58688.2023.00041>
- [68] Yeeun Yoo, Jinho Shin, and Sunghyon Kyeong. 2023. Medicare Fraud Detection Using Graph Analysis: A Comparative Study of Machine Learning and Graph Neural Networks. *IEEE Access* 11 (2023), 88278–88294. <https://doi.org/10.1109/ACCESS.2023.3305962>
- [69] Chi Zhang and Manuel Rigger. 2025. Constant Optimization Driven Database System Testing. *Proc. ACM Manag. Data* 3, 1, Article 24 (Feb. 2025), 24 pages. <https://doi.org/10.1145/3709674>
- [70] Yingying Zheng, Wensheng Dou, Lei Tang, Ziyu Cui, Yu Gao, Jiansen Song, Liang Xu, Jiaxin Zhu, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2024. Testing Gremlin-Based Graph Database Systems via Query Disassembling. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) (ISSTA 2024). Association for Computing Machinery, New York, NY, USA, 1695–1707. <https://doi.org/10.1145/3650212.3680392>
- [71] Yingying Zheng, Wensheng Dou, Lei Tang, Ziyu Cui, Jiansen Song, Ziyue Cheng, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2024. Differential Optimization Testing of Gremlin-Based Graph Database Systems. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 25–36. <https://doi.org/10.1109/ICST60714.2024.00012>
- [72] 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>
- [73] Zeyang Zhuang, Penghui Li, Pingchuan Ma, Wei Meng, and Shuai Wang. 2023. Testing Graph Database Systems via Graph-Aware Metamorphic Relations. *Proc. VLDB Endow.* 17, 4 (Dec. 2023), 836–848. <https://doi.org/10.14778/3636218.3636236>
- [74] Lei Zou, Lei Chen, and M. Tamer Özsu. 2009. DistanceJoin: Pattern Match Query In a Large Graph Database. *Proc. VLDB Endow.* 2 (2009), 886–897. <https://api.semanticscholar.org/CorpusID:5627448>