Finding Logic Bugs in Graph-processing Systems via Graph-cutting

QIUYANG MANG, The Chinese University of Hong Kong, Shenzhen, China
JINSHENG BA, ETH Zurich, Switzerland
PINJIA HE, The Chinese University of Hong Kong, Shenzhen, China
MANUEL RIGGER, National University of Singapore, Singapore

Graph-processing systems, including Graph Database Management Systems (GDBMSes) and graph libraries, are designed to analyze and manage graph data efficiently. They are widely used in applications such as social networks, recommendation systems, and fraud detection. However, logic bugs in these systems can lead to incorrect results, compromising the reliability of applications. While recent research has explored testing techniques specialized for GDBMSes, it is unclear how to adapt them to graph-processing systems in general. This paper proposes Graph-cutting, a universal approach for detecting logic bugs in both GDBMSes and various algorithms in graph libraries. Our key idea is inspired by the observation that certain graph patterns are critical for various graph-processing tasks. Dividing graph data into subgraphs that preserve those patterns establishes a natural relationship between query results on the original graph and its subgraphs, allowing for the detection of logic bugs when this relationship is violated. We implemented Graph-cutting as a tool, Gslicer, and evaluated it on 3 popular graph-processing systems, NetworkX, Neo4j, and Kùzu. Gslicer detected 39 unique and previously unknown bugs, out of which 34 have been fixed and confirmed by developers. At least 8 logic bugs detected by Gslicer cannot be detected by baseline strategies. Additionally, by leveraging just a few concrete relationships, Graph-cutting can cover over 100 APIs in NetworkX. We expect this technique to be widely applicable and that it can be used to improve the quality of graph-processing systems broadly.

CCS Concepts: • **Information systems → Data management systems**; • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Graph-processing, Graph databases, Logic bugs

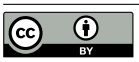
ACM Reference Format:
Qiuyang Mang, Jinsheng Ba, Pinjia He, and Manuel Rigger. 2025. Finding Logic Bugs in Graph-processing Systems via Graph-cutting. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 163 (June 2025), 27 pages. https://doi.org/10.1145/3725300

1 Introduction

Graph-processing systems, including Graph Database Management Systems (GDBMSes) and graph libraries, are designed for analyzing and handling data whose relationships are highly interconnected and can be represented as a graph. Graph library enables users to analyze graph data, *e.g.,* analyzing the importance of a web page [42], while GDBMS enables efficient querying of complex

*Qiuyang Mang worked on this project partly while visiting the National University of Singapore as a research intern.

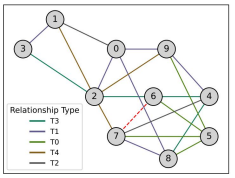
Authors' Contact Information: Qiuyang Mang, The Chinese University of Hong Kong, Shenzhen, China, qiuyangmang@link.cuhk.edu.cn; Jinsheng Ba, ETH Zurich, Switzerland, jinsheng.ba@inf.ethz.ch; Pinjia He, The Chinese University of Hong Kong, Shenzhen, China, hepinjia@cuhk.edu.cn; Manuel Rigger, National University of Singapore, Singapore, rigger@nus.edu.sg.



ACM 2836-6573/2025/6-ART163
https://doi.org/10.1145/3725300

Listing 1. Unexpected exception: runtime error in Kùzu caused by incorrect graph topology in query pattern.

```
1 CREATE NODE TABLE CITY (id INT64, PRIMARY
    KEY(id));
2 CREATE REL TABLE ROAD (FROM CITY TO CITY);
3 CREATE ({id:1})-[:R]->({id:2});
4 MATCH (a {id:1}) CREATE (a)-[:R]->(a);
5 MATCH (city1), (city2)<-[]-(city2), (city1)
    <-[]-(city2)
6 // Matching cities with a ring road and an
    intercity road
7 RETURN city1.id, city2.id;
8 --{RuntimeError: _Map_base::at}
```




```
1 gds.alpha.triangles() --{
2     line  nodeA  nodeB  nodeC
3     0       0      1      2
4                .....
5     8       5      6      7
6                .....
7     11      5      7      8
8 }
```

Fig. 1. Logic bug: incorrect triangle listing in Neo4j when the input graph contains multiple relation types.

graph data, *e.g.*, searching for specific communities in a social network [23]. These systems are critical for data-driven applications such as social network analysis [40, 50, 65] and recommendation systems [9]. For example, Neo4j [41] is one of the most popular GDBMSes according to the DB-Engines Ranking,[1] and it integrates a self-developed graph library, Neo4j-GDS. NetworkX [20] is one of the most popular graph libraries with numerous downstream applications. Kùzu [16] is a popular GDBMS developed within the research community. In addition, graph data processing has recently been standardized for SQL, with several relational DBMSes having added support for it [55].

Graph-processing systems are prone to bugs like other software, where logic bugs, crashes, and unexpected exceptions are the most typical types of bugs. Logic bugs can lead to incorrect results being returned to users, while crashes and unexpected exceptions prevent users from obtaining results. For example, Listing 1 illustrates an unexpected exception in Kùzu and Figure 1 shows a logic bug that we detected in Neo4j. In Listing 1, two cities (nodes: `id:1` and `id:2`) and two roads (relationships: `id:1` $\longrightarrow$ `id:2` and `id:1` $\longrightarrow$ `id:1`) are created in Kùzu (lines 1–4). In this graph data, Kùzu fails to retrieve results from the following query to match cities with a ring road and an intercity highway (lines 5–7). The query should retrieve $\{$`city1` $\sim$ `id:2`, `city2` $\sim$ `id:1`$\}$, but resulted in a run-time error by incorrectly computing the graph topology in the query pattern. In contrast to such unexpected exceptions, logic bugs can silently compute incorrect results, making them difficult to detect and fix. In Figure 1, Neo4j fails to return the correct result for *triangle (also known as 3-clique) listing*, which is one of the most fundamental tasks in graph processing with diverse applications [22, 43, 53, 54]. After creating the corresponding graph data with multiple relation types in Neo4j as shown in Figure 1, Neo4j-GDS lists a non-existing triangle $(5, 6, 7)$ and thus gives an incorrect triangle count of 11. The root cause of this logic bug was Neo4j improperly clearing the priority queues for the search spaces used in triangle listing when multiple relation types are present.

Testing graph-processing systems is challenging due to the lack of a universal and effective *test oracle*—a mechanism for determining whether a test case has passed or failed [6]. While previous works have proposed several test oracles for testing relational DBMSes [2, 13, 27, 29, 45, 46, 52] and GDBMSes [5, 25, 33, 36, 62, 66], to the best of our knowledge, it is still unclear how to adapt them to graph-processing systems.

A series of works on testing methodologies for relational DBMSes and GDBMSes involve generating queries, passing them to different systems or versions, and checking the equivalence of their outputs, a technique known as *differential testing* [13, 29, 33, 62]. However, differential testing in graph-processing systems often suffers from false alarms due to intentional but inconsistent

[1] https://db-engines.com/en/ranking

designs between different systems [25, 36]. For example, Kùzu [16] and Neo4j [41] will return different counts for a basic self-loop detection query, *i.e.,* `MATCH (n)-[]-(n) RETURN COUNT(*)`. Another line of work focuses on *query mutation* [5, 25, 27, 36, 45, 46, 52, 66], a technique to generate queries and their mutated versions, passing them to the same system and checking the expected relationships between their results. For example, GraphGenie [25] transforms a graph query into a new one with the injective or surjective result, *e.g.,* `MATCH (n) COUNT(*)` ⇒ `MATCH (n:user) COUNT(*)`. However, adapting query mutation-based methodologies to general graph-processing systems presents significant challenges. Specifically, (1) these methodologies focus solely on query transformation while disregarding graph structures; (2) for some graph-processing tasks, such as the triangle listing task in Figure 1, query mutation is not feasible due to the fixed input format.

To address the gap in testing graph-processing systems, we propose Graph-cutting, a high-level test oracle that can be effectively applied to both GDBMSes and graph libraries. Unlike previous approaches that focus on mutating queries, the core concept of Graph-cutting is to divide the graph data into two node sets and execute the same graph query or algorithm on both the original graph and the two subgraphs induced by these sets. While graph analysis and GDBMS queries differ across applications, a common observation is that certain subgraph structures, also known as graph patterns, play a crucial role in the analysis and processing of the underlying GDBMS and constructing intuitive models for understanding complex structures [1, 23]. Given this, most graph patterns will still appear in the two subgraphs, missing only those patterns that involve cutting edges, *i.e.,* relationships with endpoints in different node sets. This provides a fundamental relationship between the query results on the original graph and the subgraphs for various tasks in graph-processing systems.

By specifying the relationship based on the Graph-cutting oracle, we can automatically execute queries and validate the results, where any violation of these relationships indicates a logic bug in the graph-processing system. For example, the set of triangles in the original graph that do not contain any edges connecting nodes between two subgraphs (*also known as cutting edges*), should be equivalent to the union of the triangle sets within each subgraph. Based on that, we automatically detected the bug shown in Figure 1. As shown in Figure 2, we divide the nodes into two sets, $\{0, 1, 2, 3, 4, 9\}$ and $\{5, 6, 7, 8\}$. After executing the same triangle algorithm of Neo4j on the two corresponding induced subgraphs, it returns four triangles in total, $(0, 1, 2)$, $(1, 2, 3)$, $(0, 2, 9)$ and $(5, 7, 8)$. However, the triangle of $(5, 6, 7)$, whose nodes are all in the second set and which was returned in Figure 1, is missing from these four triangles, contradicting the expected relationship. Thus, Neo4j processes at least one of the three triangle listings incorrectly, revealing a logic bug in the algorithm.

Note that the process shown in Figure 1 leverages the information of cutting edges to validate the results. However, for tasks like *triangle counting*, where the information of cutting edges on results is challenging to consider, a different division approach is needed. For example, in Figure 3, we directly count the number of triangles, instead of listing them. As a result, the information on which triangles contain cutting edges is missing. To test such tasks, we need to ensure that there are no cutting edges when dividing the graphs, and thus the number of triangles in the original graph should be equivalent to the sum of those in the two subgraphs. By leveraging this new division and the oracle specification, a logic bug can be detected in triangle counting if the relationship is violated, *e.g.,* $5 \neq (1 + 3)$ in Figure 3.

After illustrating the validation of triangle counting and listing, we introduce the two divisions based on Graph-cutting: Lossy-Cutting and Lossless-Cutting. Lossy-Cutting divides the graph with cutting edges, allowing us to fully leverage the information from those edges to establish strong relationships between results, as shown in Figure 2. Lossless-Cutting, on the other hand, divides the graph without any cutting edges, *i.e.,* the two subgraphs were already disconnected in the

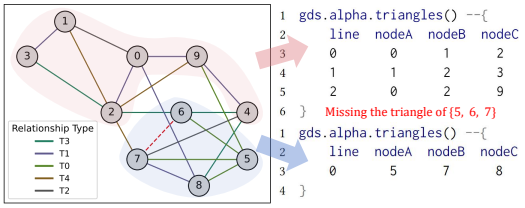


Fig. 2. An illustrative example of how Graph-cutting identifies the logic bug in Neo4j's triangle listing involves dividing the nodes into two sets: $\{0, 1, 2, 3, 4, 9\}$ and $\{5, 6, 7, 8\}$. The triangle $(5, 6, 7)$ in Figure 1 contains none of the cutting edges but is missing in both subgraph results, indicating a logic bug.

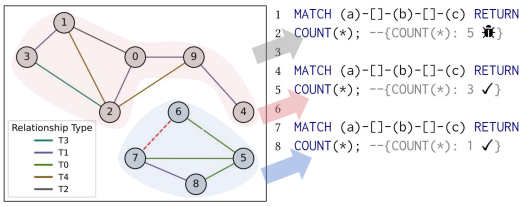


Fig. 3. An illustrative example of how Graph-cutting applies to tasks where considering the effect of cutting edges on the results is challenging. In this case, it is impossible to reduce the number of triangles containing cutting edges from the count on the original graph for validating arbitrary divisions. Instead, we remove the influence of such cutting edges by deliberately dividing them into two subgraphs that have already been disconnected in the original graph.

original graph. This provides greater flexibility for various graph-processing tasks by eliminating the influence of cutting edges, as shown in Figure 3. In this paper, we also prove that these divisions and their corresponding oracle specifications apply to all *Graph Pattern Matching* tasks [18], which can be reduced to the triangle listing that we discussed above and serve as the foundation of GDBMS querying and graph analysis [1].

We implemented Graph-cutting as a tool, Gslicer,[2] which automatically generates graph data and queries, and validates graph-processing systems. To the best of our knowledge, Graph-cutting is the first universal oracle for detecting logic bugs in general graph-processing systems, and Gslicer is the first tool designed specifically for this purpose. Using Gslicer, we can automatically identify logic bugs by checking expected relationships, as well as detect other bugs (*e.g.,* crashes, exceptions, and hangs) that impede the systems. To evaluate the effectiveness of Graph-cutting, we tested three popular and well-tested graph-processing systems, including Neo4j [41], NetworkX [20], and Kùzu [16]. As a result, we have found 39 bugs in these systems, out of which developers have confirmed 34 and 18 have been fixed. Of these confirmed or fixed bugs, 11 are previously unknown logic bugs. They are unique and were missed by existing test suites. Our work received positive feedback from the graph-processing systems' developers during the testing campaign. In addition, our experimental results show that Graph-cutting can identify bugs located in code branches covered by existing test suites, as well as 8 logic bugs that baseline strategies have overlooked. Our evaluation in NetworkX also demonstrates that already a few specifications of Graph-cutting can cover more than 100 graph-processing APIs in NetworkX, further highlighting the generalizability and effectiveness of our methodology.

In summary, we make the following main contributions:

- We propose Graph-cutting, a universal oracle to automatically validate graph-processing systems' results, and implemented an automated tool Gslicer for detecting logic bugs in graph-processing systems.
- We detected 39 previously unknown and unique bugs in three widely used graph-processing systems.
- In our evaluation, Gslicer outperforms the baseline strategies including existing test suites and previous works on GDBMS testing, and we demonstrate the generalizability of Graph-cutting both theoretically and empirically.

[2]The source code for Gslicer and for reproducing all experimental results is available at https://github.com/joyemang33/GSlicer. We also provide detailed information on the previously unknown bugs that we found.

2 Background

Graph-processing systems. Both GDBMSes and graph libraries store and manipulate data as graphs. We define the graph data used in graph-processing systems in Definition 2.1. While there may be slight variations in how different systems store graph data, these differences do not impact the effectiveness of Graph-cutting.

Definition 2.1 (Data graph). *A data graph is denoted as $G = (V, E)$, where V is a set of attributed vertices (aka nodes) and E is a set of attributed edges (aka relationships). Each edge $e \in E$ is a pair (v_i, v_j) satisfying $v_i \in V$ and $v_j \in V$.*

GDBMSes. GDBMSes play a pivotal role in graph processing, especially for graph data manipulation and supporting flexible queries. GDBMSes typically differ in their data models and query languages. In this paper, we focus on testing labeled property graph model-based GDBMSes, utilizing Cypher [17] as the query language. This model and language are widely adopted, with features supported by modern GDBMSes such as Neo4j [41], and are an important part of the recently published standard for property graph database languages, *i.e.,* the GQL standard.[3] Regardless of the differences among GDBMSes, Graph Pattern Matching (GPM) lies at their core [1, 24, 41], as it involves searching for structured graph data and returning the results, which is defined as follows.

Problem 2.2 (Graph Pattern Matching [18]). *Let two data graphs, $G_T = (V_T, E_T)$ and $G_Q = (V_Q, E_Q)$, represent the target graph and the pattern graph for querying, respectively, where G_Q is required to be self-connected (i.e., weak-connected). The task is to find all subgraphs $\mathcal{G} \subseteq G_T$ such that there exists a bijection between $\mathcal{G}$ and G_Q, satisfying graph isomorphism and matching attributes on vertices and edges.*

Specifically, most GDBMS queries can be viewed as a GPM problem. In Cypher, the `MATCH` clause declaratively describes the pattern graph G_Q and `WHERE` clauses can be regarded as the constraints on G_Q's attributes. For example, the Cypher queries in Figure 3 can be interpreted as GPM tasks, where G_T represents the graph data shown on the left, and G_Q corresponds to a 3-clique.

Graph libraries. Graph libraries provide essential tools and algorithms for the efficient analysis of graph data structures. For example, libraries like NetworkX [20] include fundamental algorithms such as depth-first search and shortest paths, alongside more specialized techniques like PageRank [42] and clustering coefficient computation [48]. Unlike GDBMSes, which use query languages, graph libraries rely on API calls to invoke specific algorithms. While these pre-designed algorithms are often more efficient for specialized graph tasks, they tend to be more limited in scope and flexibility compared to the broader query capabilities of GDBMSes. Specifically, the API call-based query format in graph libraries complicates the application of existing GDBMS testing methods, as API calls offer less flexibility for mutating queries. Consequently, how to effectively apply these GDBMS testing approaches to graph libraries remains unclear.

3 Graph-cutting

In this section, we formally define Graph-cutting, detailing two division strategies: Lossy-Cutting and Lossless-Cutting. We will then describe how to specify Graph-cutting for GPM tasks [18] and rigorously prove their correctness. GPM tasks represent a general class of graph algorithms and are a core component of GDBMSes [1]. Following this, we will extend Graph-cutting to test other important tasks in graph-processing systems, further showing its generalizability.

[3] https://jtc1info.org/slug/gql-database-language/

3.1 Definition

The key idea of Graph-cutting is to divide graph data into two subgraphs and apply the same query or algorithm to both the original graph and the two resulting subgraphs. By analyzing the relationships between the results, any inconsistencies or violations of expected relationships can reveal logic bugs in the system. In this section, we formally define Graph-cutting.

Graph-cutting on data graphs is defined as follows: Given a data graph $G = (V, E)$, we divide the vertices in G into two sets, $V^{(1)}$ and $V^{(2)}$, such that $V^{(1)} \cap V^{(2)} = \emptyset$ and $V^{(1)} \cup V^{(2)} = V$. The corresponding induced subgraphs are $G^{(1)} = (V^{(1)}, E^{(1)})$ and $G^{(2)} = (V^{(2)}, E^{(2)})$. We then define the set of cutting edges, which belong to neither of the two subgraphs, as $\mathcal{E} = \{e \mid e = (v_i, v_j) \in E$ satisfying $(v_i \in V^{(1)}, v_j \in V^{(2)})$ or $(v_i \in V^{(2)}, v_j \in V^{(1)})\}$.

Given a query Q executed on the target graph-processing system with graph data G, let $\texttt{res}(G, Q)$ represent the result returned by the system. The same query is then executed on the subgraphs, producing $\texttt{res}(G^{(1)}, Q)$ and $\texttt{res}(G^{(2)}, Q)$. As described in Definition 3.1, Graph-cutting defines the expected relationship among these three results, which should hold regardless of how the vertices are divided.

Definition 3.1 (Graph-cutting Oracle). *Given a data graph G and a division as described above, Graph-cutting is defined as:*

$$\texttt{res}(G^{(1)}, Q) \oplus \texttt{res}(G^{(2)}, Q) \preceq \mathcal{F}(\texttt{res}(G, Q), \mathcal{E}) \tag{1}$$

- *"$\oplus$" is a combination operator used to merge the results from the two subgraphs;*
- *"$\preceq$" is a comparison operator that establishes the relationship between the original and combined results; and*
- *$\mathcal{F}$ is a function that adjusts the original result by accounting for the impact of the cutting edges.*

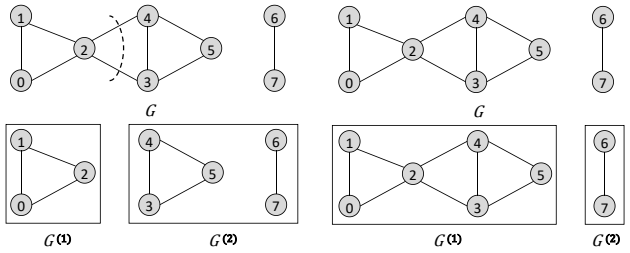
Based on the high-level definition, various specifications of Graph-cutting oracles can be applied to test different tasks in graph-processing systems. For example, when detecting the logic bug in Neo4j [41]'s triangle listing, as shown in Figure 2, the Graph-cutting oracle can be specified as follows. The vertex sets are divided into $V^{(1)} = \{0, 1, 2, 3, 4, 9\}$ and $V^{(2)} = \{5, 6, 7, 8\}$, with the cutting edges defined as

$$\mathcal{E} = \{(0, 8), (2, 6), (2, 7), (4, 5), (4, 6), (4, 7), (4, 8), (5, 9)\}.$$

In this scenario, "$\oplus$" is specified as a set union (*i.e.,* "$\cup$"), "$\preceq$" as an equality operator (*i.e.,* "$=$"), and $\mathcal{F}$ as a function that removes all triangles containing cutting edges. More formally,

$$\begin{aligned} &\texttt{res}(G^{(1)}, Q) \underbrace{\cup}_{\oplus} \texttt{res}(G^{(2)}, Q) \underbrace{=}_{\preceq} \\ &\underbrace{\texttt{res}(G, Q) \setminus \{\Delta \mid \Delta \in \texttt{res}(G, Q), \exists_{e \in \mathcal{E}} e \in \Delta\}}_{\mathcal{F}(\texttt{res}(G, Q), \mathcal{E})} \end{aligned} \tag{2}$$

This oracle indicates that any triangle found in the original graph that does not include a cutting edge should also be found in one of the subgraphs, and its correctness will be proved in Section 3.2. In the example, Neo4j fails to satisfy the oracle since $\{5, 6, 7\} \in \texttt{res}(G, Q)$ containing no cutting edges but disappear in $\texttt{res}(G^{(1)}, Q) \cup \texttt{res}(G^{(2)}, Q)$.



(a) Lossy-Cutting (b) Lossless-Cutting

Fig. 4. An illustrative example of Graph-cutting's two divisions.

Specifically, based on Graph-cutting, we propose the following two types of vertex division, as shown in Figure 4:

- Lossy-Cutting: Dividing vertices into two sets that have cutting edges in the original graph, *i.e.,* $|\mathcal{E}| > 0$. For example, Figure 4 (a) divides the graph into the two vertex sets $\{0, 1, 2\}$ and $\{3, 4, 5, 6, 7\}$ with $\mathcal{E} = \{(2, 3), (2, 4)\}$.
- Lossless-Cutting: Dividing vertices into two sets that are disconnected in the original graph, *i.e.,* $|\mathcal{E}| = 0$, and thus $\mathcal{F}$ is an identity function that $\mathcal{F}(\texttt{res}(G, \boldsymbol{Q}), \mathcal{E}) = \texttt{res}(G, \boldsymbol{Q})$. For example, Figure 4 (b) divides the graph into the two vertex sets $\{0, 1, 2, 3, 4, 5\}$ and $\{6, 7\}$ with no cutting edges.

Intuitively, Lossy-Cutting provides a relatively strong test oracle by fully accounting for the impact of cutting edges, whereas Lossless-Cutting offers more flexibility for tasks where calculating the effect of cutting edges is challenging. We will elaborate on this in Section 3.2.

3.2 Graph-cutting for GPM

Many important tasks in graph-processing systems can be viewed as GPM or its variations, such as the MATCH query in GDBMSes [38, 67], and the motif-counting [8] in social network analysis [44]. Specifically, the triangle listing task in Figure 1 can be regarded as a GPM problem with G_Q being a 3-clique. Despite being foundational to GDBMSs and having widespread applications, there is no unified mechanism to effectively verify the implementations of GPM-based problems due to their inherent **NP**-hard complexity [18].

We will show how Graph-cutting holds in GPM-based tasks. First, when G_Q is connected, we have the following specification for GPM based on Lossy-Cutting.

Theorem 3.1 (Lossy-Cutting for GPM). *Given a GPM query with $G_T = (V_T, E_T)$ and $G_Q = (V_Q, E_Q)$ satisfying G_Q is connected, let $\texttt{res}(G_T, G_Q)$ denote the result of the GPM query. Consider any two subgraphs of G_T derived by the Lossy-Cutting, $G_T^{(1)} = (V_T^{(1)}, E_T^{(1)})$ and $G_T^{(2)} = (V_T^{(2)}, E_T^{(2)})$, and let $\mathcal{E}$ be the set of cutting edges. We have:*

$$\begin{aligned}&\texttt{res}(G_T^{(1)}, G_Q) \cup \texttt{res}(G_T^{(2)}, G_Q) = \texttt{res}(G_T, G_Q) \setminus \\ &\{\mathcal{G} \mid \mathcal{G} \in \texttt{res}(G_T, G_Q), \exists_e\, e \in \mathcal{E} \wedge e \in \mathcal{G}\}\end{aligned} \tag{3}$$

To see this, we need to prove the following lemma.

Lemma 3.2. *For any subgraph $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G})$ of $\texttt{res}(G_T, G_Q)$, there exists a cutting edge $e \in \mathcal{G}$, if and only if $V_\mathcal{G} \cap V_T^{(1)}$ and $V_\mathcal{G} \cap V_T^{(2)}$ are both non-empty.*

Proof. **(1)** If a cutting edge e exists, its two endpoints must belong to two different vertex sets. **(2)** If $V_{\mathcal{G}}$ contains at least one vertex from $V_T^{(1)}$ and at least one from $V_T^{(2)}$, without any cutting edge in $\mathcal{G}$ would disconnect $\mathcal{G}$. Since there is an isomorphism between $\mathcal{G}$ and G_Q, this contradicts the fact that G_Q is connected.

□

Based on Lemma 3.2, the subgraphs $\mathcal{G}$ in $\texttt{res}(G_T, G_Q)$ that contain no cutting edges are also subgraphs of $G_T^{(1)}$ or $G_T^{(2)}$. This means they will be included in $\texttt{res}(G_T^{(1)}, G_Q)$ or $\texttt{res}(G_T^{(2)}, G_Q)$ exactly once, since $V_T^{(1)} \cap V_T^{(2)} = \emptyset$ and $V_T^{(1)} \cup V_T^{(2)} = V_T$. Additionally, since $G_T^{(1)}$ and $G_T^{(2)}$ are subgraphs of G_T, the same GPM query will only retrieve subgraphs already present in $\texttt{res}(G_T, G_Q)$, which finally proves Theorem 3.1.

Theorem 3.1 demonstrates that Lossy-Cutting can be used to test various GPM problems, regardless of attribute matching and the structure of the query pattern, as long as connectivity is ensured. Meanwhile, the following extended relationships can be adapted to related tasks, such as *Graph Pattern Counting*, where we only want to determine the size of $\texttt{res}(G_T, G_Q)$.

$$\texttt{res}(G_T^{(1)}, G_Q) \cup \texttt{res}(G_T^{(2)}, G_Q) \subseteq \texttt{res}(G_T, G_Q) \text{ and,} \tag{4}$$

$$|\texttt{res}(G_T^{(1)}, G_Q)| + |\texttt{res}(G_T^{(2)}, G_Q)| \leq |\texttt{res}(G_T, G_Q)| \tag{5}$$

However, when G_Q is not connected, we use Lossless-Cutting, as stated in the following theorem.

Theorem 3.2 (Lossless-Cutting for GPM). *Given a GPM query with $G_T = (V_T, E_T)$ and $G_Q = (V_Q, E_Q)$. Consider any two subgraphs of G_T derived by Lossless-Cutting, $G_T^{(1)} = (V_T^{(1)}, E_T^{(1)})$ and $G_T^{(2)} = (V_T^{(2)}, E_T^{(2)})$. We have*

$$\texttt{res}(G_T^{(1)}, G_Q) \cup \texttt{res}(G_T^{(2)}, G_Q) \subseteq \texttt{res}(G_T, G_Q) \tag{6}$$

Proof. When G_Q is connected, the theorem can be directly derived from Theorem 3.1. Since there are no cutting edges between $G_T^{(1)}$ and $G_T^{(2)}$, we have $\texttt{res}(G_T^{(1)}, G_Q) \cup \texttt{res}(G_T^{(2)}, G_Q) = \texttt{res}(G_T, G_Q)$.

Otherwise, we consider each maximal connected subgraph $\mathcal{G}$ in $\texttt{res}(G_T, G_Q)$. Each must be a subgraph of either $G_T^{(1)}$ or $G_T^{(2)}$. For all Gs where all subgraphs belong to one side, there is a bijection between them and $\texttt{res}(G_T^{(1)}, G_Q) \cup \texttt{res}(G_T^{(2)}, G_Q)$, indicating that $\texttt{res}(G_T, G_Q)$ is a proper superset of the two subgraphs' results. □

Similar to Lossy-Cutting, we also have the following extension for Lossless-Cutting:

$$|\texttt{res}(G_T^{(1)}, G_Q)| + |\texttt{res}(G_T^{(2)}, G_Q)| \leq |\texttt{res}(G_T, G_Q)| \tag{7}$$

Note that when G_T is self-connected, it will be impossible to find a non-trivial cut for Lossless-Cutting, meaning both $V_T^{(1)}$ and $V_T^{(2)}$ will be non-empty. However, in our implementation, we can manually create at least two disconnected components when generating graph data (see Section 4).

Recall that most GDBMS queries can be viewed as GPM tasks. These theorems offer theoretical guarantees when applying Graph-cutting to such queries. Importantly, during GDBMS testing, we generate only queries that qualify as GPM tasks.

3.3 Other Tasks

In addition to GPM-based tasks that directly query specific subgraph structures, we also extend Graph-cutting to other important tasks in graph-processing systems. As shown in Table 1,

Table 1. Specifications of Graph-cutting for representative tasks in graph-processing systems.

Category	Task	Divisions	Combination Operator $\oplus$	Comparison Operator $\preceq$	Adjusting Function $\mathcal{F}$
Centrality	PageRank	Lossless	$\cup$[1]	= (Pairwise ranking)[3]	$\mathcal{I}$[2]
	Harmonic Centrality	Lossless	$\cup$	=[4]	$\mathcal{I}$
Community Detection	K-core Decomposition	Lossy	$\cup$	$\leq$[5]	$\mathcal{I}$
		Lossless	$\cup$	=	$\mathcal{I}$
	Strongly Connected Components	Lossy	$\cup$	$\supseteq$	Excluding all components containing cutting edges $\in \mathcal{E}$
		Lossless	$\cup$	=	$\mathcal{I}$
Path Finding	All Pairs Shortest Path	Lossy	$\cup$	$\geq$[6]	$\mathcal{I}$
	Minimum Spanning Tree (positive weights)	Lossy	$\cup$	$\geq$ (sum of edge weights)[7]	Excluding all cutting edges $\in \mathcal{E}$ in MST
Similarity	Node Similarity	Lossless	$\cup$	=	$\mathcal{I}$

[1] Combining two result sets by $\texttt{res}(G^{(1)}) \cup \texttt{res}(G^{(2)})$.
[2] Identity function, $\mathcal{F}(x) = x$.
[3] For each subgraphs $G^{(i)}, i \in \{1, 2\}$ and each pair of vertices $(v_1, v_2) \in G^{(i)}$ satisfying $\texttt{rank}(v_1) < \texttt{rank}(v_2) \in \texttt{res}(G^{(i)})$, checking whether $\texttt{rank}(v_1) < \texttt{rank}(v_2)$ still holds in $\texttt{res}(G)$, *i.e.,* $\texttt{rank}(v_1) < \texttt{rank}(v_2) \in \texttt{res}(G^{(i)}) \Rightarrow \texttt{rank}(v_1) < \texttt{rank}(v_2) \in \texttt{res}(G)$.
[4] Checking whether two result sets are equivalent.
[5] For two key-value result sets A, B, for each key k with corresponding value v_A and v_B in the two sets, checking whether $v_A \leq v_B$.
[6] For two key-value result sets A, B, for each key k with corresponding value v_A and v_B in the two sets, checking whether $v_A \geq v_B$.
[7] For two weighted edge result sets A, B, checking whether $\sum_{(u,v,w)\in A} w \geq \sum_{(u,v,w)\in B} w$.

we selected 1–2 representative tasks from Neo4j's various graph algorithm categories, including *PageRank*, *Harmonic Centrality*, *K-core Decomposition*, *Strongly Connected Components*, *Shortest Path*, *Minimum Spanning Tree*, and *Node Similarity*.[4]

Although these tasks differ in their algorithms, queries, result formats, and applications, they all implicitly rely on key subgraph structures, which enables us to define Graph-cutting specifications for them.

We now present several examples with proof sketches. These serve as representative examples of how specific Graph-cutting oracles can be proven correct. Take the shortest path task as the first example, which is defined as follows:

Problem 3.3 (All Pairs Shortest Path (APSP)[5]). *Given a weighted, undirected graph G, return a 2D table $\texttt{res}(G)$ such that for each pair of vertices $(u, v) \in G$, $\texttt{res}(G)[u][v]$ represents the length of the shortest path between u and v.*

Given a graph G and its dividing subgraphs $G^{(1)}$ and $G^{(2)}$, the Lossy-Cutting for APSP, as shown in Table 1, states that for every pair of vertices (u, v) that are either both in $G^{(1)}$ or both in $G^{(2)}$, the length of the shortest path between them in the respective subgraph is no less than the length of which in the original graph. *i.e.,* either $\texttt{res}(G^{(1)})[u][v] \geq \texttt{res}(G)[u][v]$ or $\texttt{res}(G^{(2)})[u][v] \geq \texttt{res}(G)[u][v]$.

Proof sketch. Since all paths between any pair of vertices u and v in the dividing subgraphs will remain in the original graph, it is easy to see that the correctness of the oracle holds. □

[4] https://neo4j.com/docs/graph-data-science/current/algorithms/
[5] https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.shortest_paths.unweighted.all_pairs_shortest_path.html

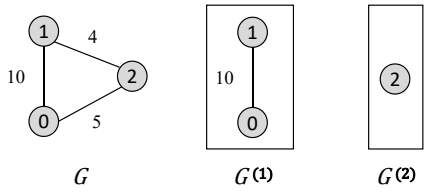


Fig. 5. An illustrative example of Graph-cutting for the Shortest Path problem, where the shortest path length between each pair in the original graph should not exceed that in the subgraphs, *e.g.,* the shortest path length between nodes 1 and 2 is 9 in G and 10 in $G^{(1)}$.

For instance, as shown in Figure 5, the length of the shortest path between vertex 1 and vertex 2 in $G^{(1)}$ must be no less than the length in G. This is because intermediate vertices, such as vertex 3, which may be part of the shortest path in G, could be absent in the subgraphs.

For another example, the task of Strongly Connected Components is defined as follows.

Problem 3.4 (Strongly Connected Components[6]). *Given a directed graph G, return* $\texttt{res}(G)$ *as the set of all Strongly Connected Components (SCCs) from G, where SCC is a maximal subset of vertices where each vertex in the subset is reachable from every other vertex.*

Lossless-Cutting. Given a graph G and its dividing subgraphs $G^{(1)}$ and $G^{(2)}$, the Lossless-Cutting oracle shown in Table 1 states that, $\texttt{res}(G) = \texttt{res}(G^{(1)}) \cup \texttt{res}(G^{(2)})$ if there is no cutting edge in G connecting two vertices belonging to $G^{(1)}$ and $G^{(2)}$, respectively.

Proof sketch. We first show that, for each SCC $H \in \texttt{res}(G)$, either $H \in \texttt{res}(G^{(1)})$ or $H \in \texttt{res}(G^{(2)})$. Since there is no cutting edge, H cannot simultaneously contain vertices from both $G^{(1)}$ and $G^{(2)}$, as this would prevent the vertices belonging to different subgraphs from reaching each other. Hence, H is either a subgraph of $G^{(1)}$ or $G^{(2)}$. In addition, H is still maximal in $G^{(1)}$ and $G^{(2)}$ as they are subgraphs of G, which means that H is either in $\texttt{res}(G^{(1)})$ or $\texttt{res}(G^{(2)})$. Similarly, we can prove the converse, *i.e.,* for each $H \in \texttt{res}(G^{(1)})$ or $H \in \texttt{res}(G^{(2)})$, $H \in \texttt{res}(G)$. Therefore, we have $\texttt{res}(G) = \texttt{res}(G^{(1)}) \cup \texttt{res}(G^{(2)})$. □

Lossy-Cutting. The Lossy-Cutting oracle shown in Table 1 for SCC states that, after removing all SCC containing cutting edges from $\texttt{res}(G)$, it will become a subset of $\texttt{res}(G^{(1)}) \cup \texttt{res}(G^{(2)})$.

Proof sketch. Note that for each SCC H in $\texttt{res}(G)$ that does not contain any cutting edges, it can only include vertices from one side of $G^{(1)}$ or $G^{(2)}$. Therefore, we only need to prove that those SCCs are either in $\texttt{res}(G^{(1)})$ or $\texttt{res}(G^{(2)})$, which we have proven in the proof of the correctness of Lossless-Cutting for SCC. □

Lastly, we show how Graph-cutting works for Minimum Spanning Tree:

Problem 3.5 (Minimum Spanning Tree (MST) [7]). *Given an undirected and positive weighted graph G, return* $\texttt{res}(G)$ *as a set of edges in a minimum spanning tree or forest on G.*

Given a graph G and its dividing subgraphs $G^{(1)}$ and $G^{(2)}$, the Lossy-Cutting for MST, as shown in Table 1, states that, after excluding all cutting edges in $\texttt{res}(G)$, the weight of the minimum spanning tree or forest on G is no greater than the sum of weights on subgraphs.

[6]https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.components.strongly_connected_components.html

[7]https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.tree.mst.minimum_spanning_tree.html

Proof sketch. After excluding all cutting edges in $\texttt{res}(G)$, we only have two types of edges in the MST, connecting vertices that are either both in $G^{(1)}$ or both in $G^{(2)}$. We denote the two types of edges as $E^{(1)}$ and $E^{(2)}$, respectively. After that, we need to prove that, the sum of edge weights over $E^{(1)}$ and $E^{(2)}$ is no greater than which over $\texttt{res}(G^{(1)})$ and $\texttt{res}(G^{(2)})$. We can show that

$$\sum_{(u,v,w)\in E^{(1)}} w \leq \sum_{(u,v,w)\in \texttt{res}(G^{(1)})} w,$$
$$\sum_{(u,v,w)\in E^{(2)}} w \leq \sum_{(u,v,w)\in \texttt{res}(G^{(2)})} w,$$

where w represents the edge weight.

First, if $E^{(1)}$ is a spanning tree of $G^{(1)}$, its weight cannot be greater than the weight of $\texttt{res}(G^{(1)})$; otherwise, we could replace $E^{(1)}$ with $\texttt{res}(G^{(1)})$ in $\texttt{res}(G)$ to obtain a spanning tree of G with a smaller weight. Conversely, if $E^{(1)}$ is not a spanning tree of $G^{(1)}$, we can still replace $E^{(1)}$ with $\texttt{res}(G^{(1)})$ in $\texttt{res}(G)$. Although this will no longer form a spanning tree due to the presence of redundant edges, we can repeatedly remove one edge from each cycle until the remaining set of edges forms a spanning tree. Since all edge weights are positive, this process can only reduce the weight, ultimately resulting in a spanning tree of G with a smaller weight. The proof for $E^{(2)}$ and $\texttt{res}(G^{(2)})$ is similar, and we ultimately prove the correctness of the oracle by summing the two inequalities. □

4 Approach

Based on Graph-cutting, we implemented an automated testing approach and a tool for graph-processing systems, Gslicer, which includes three key components, the *Graph Generator*, the *Query Generator*, and the *Results Validator*.

Figure 6 provides an illustrative example of Gslicer's workflow. Initially, we generate three graphs in the target graph-processing system, G, $G^{(1)}$, and $G^{(2)}$, using our graph generator, as well as pass the cutting edges set $\mathcal{E}$ to the downstream components. The process is guided by the division strategies that we will apply, determining whether the original graph G is split into subgraphs $G^{(1)}$ and $G^{(2)}$, or synthesize G from $G^{(1)}$ and $G^{(2)}$. The query generator automatically creates syntactically valid queries for the three graphs. For GDBMSes, we use a syntax-based generator to produce Cypher queries [17], while for graph libraries, we create API calls by parsing the interface and documentation. Meanwhile, based on the query and division, we apply a test oracle based on Graph-cutting to the result validator, which can be manually specified or supplied by an automated mechanism that we present in Section 4.3. After obtaining the query results from the three graphs, the validator checks whether the relationships between the results and the set of cutting edges $\mathcal{E}$ satisfy the expected oracle. Any violation indicates a logic bug in the target system, while other types of bugs may be detected if the system fails to return results.

4.1 Graph Generation

Gslicer first generates a shared schema for the three graphs, including labels, data types of properties, and keys. Based on this schema, the graph generator randomly creates nodes and relationships with valid labels and properties.

For Lossy-Cutting, we begin by creating the graph $G = (V, E)$ using the generator, and then randomly divide the nodes in G into two subsets, denoted as $V^{(1)}$ and $V^{(2)}$. From these subsets, the subgraphs $G^{(1)}$ and $G^{(2)}$ are created by copying the nodes and retaining only the relationships within each subset, *i.e.,*

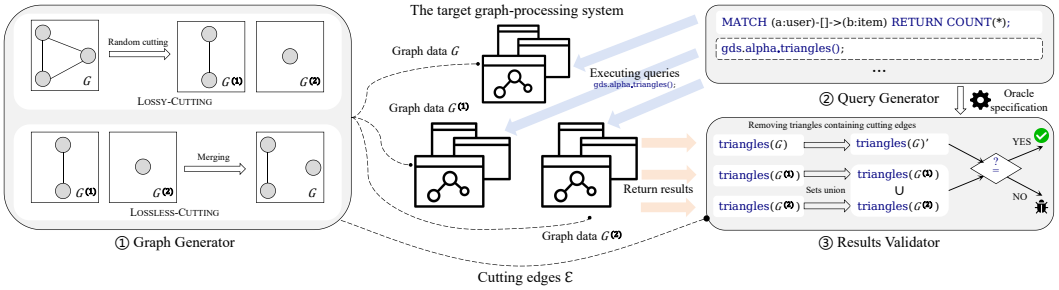


Fig. 6. An overview of Gslicer

Listing 2. Core grammar of Cypher queries [33]

```
query ::= RETURN ret | clause query
ret ::= * | expr [AS a] | ret, expr [AS a]
clause ::= [OPTIONAL] MATCH pattern [WHERE expr]
         | WITH ret [WHERE expr]
pattern ::= NODE | NODE REL pattern
```

$$G^{(1)} = (V^{(1)}, E^{(1)} = \{e = (u, v) \mid e \in E, u, v \in V^{(1)}\}),$$
$$G^{(2)} = (V^{(2)}, E^{(2)} = \{e = (u, v) \mid e \in E, u, v \in V^{(2)}\}).$$

The set of relationships that are not preserved in $G^{(1)}$ and $G^{(2)}$ forms the cutting edges set $\mathcal{E}$, *i.e.*, $\mathcal{E} = E \setminus (E^{(1)} \cup E^{(2)})$.

For Lossless-Cutting, the above strategy may fail to generate the expected graphs, as the random division of vertices does not guarantee that no cutting edges exist between the two subgraphs (*i.e.*, $\mathcal{E} = \emptyset$), and it is impossible to find a Lossless-Cutting if the original graph G is already self-connected. Instead, we first generate two random graphs, $G^{(1)}$ and $G^{(2)}$, under the same schema. We then create G by merging all nodes and relationships from $G^{(1)}$ and $G^{(2)}$. This approach ensures a valid Lossless-Cutting with no cutting edges between $G^{(1)}$ and $G^{(2)}$.

4.2 Query Generation

Despite sharing underlying implementations in the graph-processing systems, Gslicer uses different strategies for GDBMS and graph library queries because GDBMS queries rely on query language syntax, while graph library queries are executed through API calls.

GDBMS query. A GDBMS query can generally be viewed as a specialized GPM task, for which we have theoretically demonstrated the correctness of Graph-cutting in Section 3.2. Consequently, Gslicer can be query-agnostic in the general context of GDBMSes, with the only requirement being that the generated query must be deterministic. We follow the official grammar to automatically generate queries like other works [25, 33, 36, 62].

In this work, we focus on the most widely used GDBMS query language, Cypher [17], whose core syntax rules are outlined in Listing 2. Based on that, we implemented a query generator by iteratively appending clauses with random graph patterns and `WHERE` predicates. In addition to our query generator, some orthogonal works [2, 56], aiming to generate queries with higher efficiency and complexity, can be combined with Gslicer to improve the testing efficiency further.

Graph library query. Unlike GDBMS queries, which are specified using query languages, graph library queries are typically API calls that take graph data and optional parameters as inputs. For

Algorithm 1: Automated oracle specification for graph libraries

Input: an API for the target graph task: $\mathcal{A}$, a set of candidate specifications of Graph-cutting: O, and a threshold τ;
Output: a set of potential valid specifications: $O_{\mathcal{A}} \subseteq O$;

$O_{\mathcal{A}} \leftarrow \emptyset$
for *each oracle specification* $(\oplus, \preceq, \mathcal{F}) \in O$ **do**
  $cnt \leftarrow 0$;
  while $cnt < \tau$ **do**
    $G, G^{(1)}, G^{(2)}, \mathcal{E} \leftarrow$ `GraphGenerator(); // small-scale`
    $\texttt{res}_G \leftarrow \mathcal{F}(\mathcal{A}(G), \mathcal{E})$;
    $\texttt{res}_{G^{(1)} \oplus G^{(2)}} \leftarrow \mathcal{A}(G^{(1)}) \oplus \mathcal{A}(G^{(2)})$;
    if not $(\texttt{res}_{G^{(1)} \oplus G^{(2)}} \preceq \texttt{res}_G)$ **then break** ;
    $cnt \leftarrow cnt + 1$;
    if $cnt = \tau$ **then** $O_{\mathcal{A}} \leftarrow O_{\mathcal{A}} \cup \{(\oplus, \preceq, \mathcal{F})\}$;
return $O_{\mathcal{A}}$;

example, NetworkX [20] provides an API for eigenvector_centrality(G, *weight* = None, *max_iter* = 50, *tol* = 0), where the graph object (*i.e., G*) and three optional parameters (*i.e., weight*, *max_iter*, and *tol*) with default values are passed as input. Additionally, some APIs may require the input graph to meet specific conditions, such as being directed or having no duplicate edges. To efficiently generate valid queries for graph libraries, we parse the library's interface and documentation, identifying the input graph requirements and leaving other optional parameters at their default values.

4.3 Test Oracles

GDBMS query. For GDBMS queries, we can specify the two oracles detailed in Section 3.2 for all generated queries with minimal engineering effort. Specifically, Gslicer generates queries with self-connected patterns for Lossy-Cutting and all queries for Lossless-Cutting. For Lossy-Cutting, we use the specifications outlined in Theorem 3.1, while for Lossless-Cutting, we apply the specifications in Theorem 3.2. In cases where Cypher queries involve aggregating functions (line 2 of Listing 2), such as `COUNT()` or `MIN()`, Gslicer generalize the oracles by replacing the combination operator $\oplus$ with the corresponding operator used by the aggregating function. For example, GDBMS queries (1) `MATCH (n) RETURN COUNT(*)` and (2) `MATCH (n) RETURN MIN(n.id)` would be checked by the following two oracles, respectively:

$$\texttt{res}(G^{(1)}, G_Q) + \texttt{res}(G^{(2)}, G_Q) \leq \texttt{res}(G, G_Q),$$
$$\min\{\texttt{res}(G^{(1)}, G_Q), \texttt{res}(G^{(2)}, G_Q)\} \geq \texttt{res}(G, G_Q).$$

Graph library query. For graph library queries, in addition to manually specifying Graph-cutting oracles, *e.g.,* the specifications shown in Table 1, we also design a mechanism to identify potential oracle specifications automatically. This approach can help developers reduce their engineering effort, especially since graph libraries can contain hundreds of APIs.

The key idea behind our automated identification process is as follows: given an Graph-cutting oracle $(\oplus, \preceq, \mathcal{F})$, if a target graph task successfully passes this oracle with numerous small-scale graph data, *i.e.,* $\texttt{res}(G^{(1)}, Q) \oplus \texttt{res}(G^{(2)}, Q) \preceq \mathcal{F}(\texttt{res}(G, Q), \mathcal{E})$ consistently holds for these exploratory datasets, it is highly likely to be a valid specification. In addition, due to the small scale of the data, our intuition is that logic bugs are less likely to be triggered during the exploration

process. This specification can then be applied to test larger-scale graph data and future versions of graph-processing systems.

As shown in Algorithm 1, we construct a candidate set of specifications $\mathcal{O}$ by manually enumerating operators, and try to select a subset of specifications $\mathcal{O}_{\mathcal{A}}$ that can be applied to the target API $\mathcal{A}$. We first initialize $\mathcal{O}_{\mathcal{A}}$ as an empty set (line 1). We then enumerate all candidate specifications $(\oplus, \preceq, \mathcal{F})$ from $\mathcal{O}$ (line 2). For each specification, we repeat the following exploratory process τ times, controlled by the variable *cnt* (line 4, 3, and 9). Each time, we start by randomly generating three small-scale graphs $G, G^{(1)}, G^{(2)}$ with the cutting edge sets $\mathcal{E}$ using our graph generator (line 5). Note that, these graphs will satisfy the division requirements of the oracle specification, *i.e.,* Lossy-Cutting or Lossless-Cutting. Subsequently, we pass the graph G to $\mathcal{A}$ to obtain the result, which is then adjusted by the function $\mathcal{F}$ to produce $\texttt{res}_G$ (line 6). We also pass the graphs $G^{(1)}$ and $G^{(2)}$ to $\mathcal{A}$ to obtain their results, which are combined using $\oplus$ to produce $\texttt{res}_{G^{(1)} \oplus G^{(2)}}$ (line 7). Finally, we use $\preceq$ to verify the expected relationship of the specification, and stop the validation process if the check fails (line 8). We consider $(\oplus, \preceq, \mathcal{F})$ as a potential specification for $\mathcal{A}$ if the expected relationship holds for all τ trials, and then add it to $\mathcal{O}_{\mathcal{A}}$ (line 10).

Designing or identifying an appropriate adjusting function $\mathcal{F}$ may be difficult for certain tasks. One possible solution is to use Lossless-Cutting for certain complex tasks. Another approach is to design a weaker comparator and avoid using an adjusting function altogether. For example, in the triangle counting task, we can validate the result as $\texttt{res}(G, Q) \geq \texttt{res}(G^{(1)}, Q) + \texttt{res}(G^{(2)}, Q)$, replacing the equivalence comparator used in the triangle-listing task with a weaker one.

After applying the Graph-cutting oracle, the result validator will verify whether the expected relationship holds among the three returned results. Any violation indicates a logic bug.

5 Evaluation

In this section, we empirically evaluate Graph-cutting's ability to find unknown bugs, compare it with state-of-the-art approaches, and analyze its coverage of code and applicable tasks.

5.1 Discovering Unknown Bugs

Target graph-processing systems. We evaluate our approach in three mature and actively maintained graph-processing systems: Neo4j [41], NetworkX [20], and Kùzu [16]. Specifically, Neo4j is the most widely-used GDBMS according to the DB-Engine Ranking and includes its own graph library, Neo4j-GDS, built on the same underlying system. We evaluate Gslicer on both Neo4j's GDBMS and Neo4j-GDS. Additionally, we assess Gslicer on NetworkX, one of the most popular graph libraries, and Kùzu, an embedded GDBMS actively maintained by both research and commercial communities.

Overview results. Table 2 summarizes our evaluation results on discovering unknown bugs in these systems. We categorize our detected unique bugs into the following four types according to developers' feedback.

- *Fixed bugs* refer to those previously unknown bugs that developers have confirmed and addressed with fixing patches after our bug reports.
- *Confirmed bugs* refer to those previously unknown bugs that developers have confirmed in our bug report, but have not yet been fixed.
- *Unconfirmed bugs* refer to those we reported, but developers were unwilling to fix them.
- *Duplicate bugs* refer to bugs with the same confirmed root cause as bugs in our previous reports.

Overall, we reported 39 issues, as shown in Table 2. Of these, 34 have been confirmed as previously unknown and unique bugs, with 18 having already been fixed by developers. Table 3 further categorizes those confirmed or fixed bugs, 11 of which are previously unknown logic bugs. The

Table 2. Bugs status in graph-processing systems.

Graph-processing system	Fixed	Confirmed	Duplicate	Unconfirmed	Sum
Neo4j	3	0	1	0	4
NetworkX	10	12	1	2	25
Kùzu	5	4	1	0	10
Sum	18	16	3	2	39

Table 3. Classification of fixed or confirmed bugs.

Graph-processing system	Logic bugs	Crashes / Exceptions	Hang bugs	Sum
Neo4j	2	1	0	3
NetworkX	4	18	0	22
Kùzu	5	3	1	9
Sum	11	22	1	34

remaining bugs are crashes, unexpected exceptions, or hangs, which were detected when the graph-processing systems failed to return results when executing queries for Graph-cutting. For the remaining 5 reports, two were marked unconfirmed due to floating-point issues or insufficient input validation, which developers decided not to address. The other three were duplicates, as a single issue affected multiple APIs in graph-processing systems.

Developers' feedback. Developer feedback is a key indicator of the real-world impact of testing efforts, and we received a significant amount of positive responses from developers. In addition to receiving numerous acknowledgments like "*Thank you for bringing this issue to our attention*", Neo4j's developers also invited us to test their upcoming release following a patch to fix the identified issues.[8] NetworkX's developers also emphasized our work in finding several logic bugs with corner cases "*We very much appreciate the work you're doing by finding & reporting these cases! Out of curiosity - are you using some sort of 'fuzzing' tool or otherwise procedural workflow to find corner cases?*"[9] In our testing campaign on Kùzu, one of the logic bugs we identified was labeled as *high-priority*.[10] Out of more than 4,000 historical issues in Kùzu, only 65 (~2%) have received this label, highlighting the significance of the bug we detected.

Illustrative cases of logic bugs. We illustrate some interesting bugs detected by Gslicer and have already been fixed by developers, to better understand the bug patterns and demonstrate how Graph-cutting effectively identifies them.

Integer overflow in Neo4j. Listing 3 shows a logic bug in Neo4j's GDBMS caused by improper handling of integer overflows. In this case, we created a graph with 128 nodes (lines 1–4) and counted the number of 9-tuples `n0`, `n1`, . . ., `n8` in the graph (lines 5–7), which should return 2^{63} as a result. However, since Neo4j used the basic `LongLong` data type to store the counting result, an overflow occurred, resulting in a return value of -2^{63}. Graph-cutting can detect this bug using Lossless-cutting to divide the graph into two subgraphs with 64 nodes. The same query in the subgraphs can correctly return 2^{54}. By using the specification in Equation 7, the relationship has been violated by $2^{54} + 2^{54} < -2^{63}$, revealing the logic bug.

[8]https://github.com/neo4j/graph-data-science/issues/281
[9]https://github.com/networkx/networkx/issues/6913
[10]https://github.com/kuzudb/kuzu/issues/3570

Listing 3. Integer overflow when Neo4j counting the pattern with large result.

```
CREATE ({id:1});
CREATE ({id:2});
...
CREATE ({id:128}); // Creating a graph with 128 nodes
MATCH (n0) MATCH (n1) MATCH (n2) MATCH (n3)
MATCH (n4) MATCH (n5) MATCH (n6) MATCH (n7)
MATCH (n8) RETURN COUNT(*); --{-9223372036854775808} 🐞
```

Listing 4. Incorrect calculation of effective size for isolated nodes in NetworkX.

```
import networkx as nx
G = nx.Graph()
G.add_node(0), G.add_node(1), G.add_node(2)
G.add_edge(0, 1), G.add_edge(1, 2)
nx.effective_size(G)  --{0: 1.0, 1: 2.0, 2: 1.0} ✓
G1 = nx.Graph()
G1.add_node(0), G.add_node(1)
G1.add_edge(0, 1)
nx.effective_size(G1) --{0: 1.0, 1: 1.0} ✓
G2 = nx.Graph()
G2.add_node(2)
nx.effective_size(G2) --{2: nan} 🐞
```

NaN results returned in NetworkX's effective size. In NetworkX, nx.effective_size(G) calculates the effective size [7] of nodes' ego networks in a given graph G, which measures the extent to which a node's connections are non-redundant. It is defined as follows for each node u in a graph.

$$e(u) = \sum_{v \in N(u) \setminus \{u\}} (1 - \sum_{w \in N(v)} p_{uw} m_{vw}),$$

where $N(u)$ is the set of neighbors of vertex u, p_{uw} is the normalized mutual weight of the edges between vertices u and w, and m_{vw} is the mutual weight of vertices v and w. The normalized mutual weight p_{uw} is the mutual weight between u and w divided by the highest mutual weight u has with any of its neighbors. The mutual weight of v and w is the sum of the weights of the edges connecting them. It reflects the diversity of a node's neighbors by considering the degree of overlap among them, providing insight into how much unique information or resources the node can access through its network.

As shown in Listing 4, we detected a logic bug in the calculation of effective size in NetworkX by Lossy-Cutting. Specifically, we first created an original graph G with a 3-node chain $0 \leftrightarrow 1 \leftrightarrow 2$ and used Lossy-Cutting to divide it into two subgraphs: $G^{(1)} : 0 \leftrightarrow 1$ and $G^{(2)} : 2$. Since cutting only reduces the size of ego networks, the effective size should also decrease for each node. However, in subgraph $G^{(2)}$, where node 2 is isolated and its effective size should be 0 according to the definition, nx.effective_size(G2) incorrectly returned nan, violating the Graph-cutting oracle. The bug stemmed from NetworkX's simplified equation for calculating the effective size in unweighted graphs, $n - \frac{t}{n}$, where t refers to the number of edges in the induced graph of $N(u)$, and $n = |N(u)| - 1$. This equation missed the case when $N(u) = 1$ by returning an incorrect nan.

Incorrect large number returned in NetworkX's trophic level. In NetworkX, nx.trophic_level(G) in NetworkX calculates the trophic level [32] of each node in a directed graph G. The trophic level is a measure of a node's position in a hierarchical food web or flow network. In this context, basal nodes (nodes with no incoming edges) have a trophic level of 1, and non-basal nodes have a trophic level determined by the average trophic level of their predecessors, incremented by 1. The function requires all nodes in the graph can be reached by at least one basal node. As

Listing 5. Incorrect path checking from each node to a basal node in trophic level calculations in NetworkX, mistakenly returning random large numbers.

```
import networkx as nx
edges = [(0, 1), (0, 2), (0, 3), (4, 3), (5, 1),
(5, 6), (1, 5), (1, 7), (1, 8), (9, 5),
(9, 3), (10, 6), (10, 11), (10, 12), (13, 6),
(14, 2), (14, 15), (6, 9), (6, 10), (11, 13),
(16, 14), (17, 0), (17, 10), (17, 13), (12, 17)]
G = nx.DiGraph(edges)
nx.trophic_levels(G)
--{4: 1 ✓, 16: 1 ✓, 0: 2.3804740887529764e+16 🐞, ...}
```

Listing 6. Incorrect pattern matching in Kùzu caused by erroneous planning of Worst-Case-Optimal Join

```
CREATE NODES, RELS from tinysnb dataset
MATCH (a:person)-[:knows]->(b:person),
(b)-[:knows]->(a), (a)-[:knows]->(b)
RETURN COUNT(*) --{16} 🐞 --{12} ✓
```

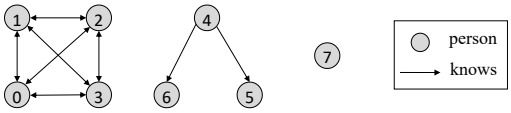


Fig. 7. The visualization of the tinysnb dataset used in Listing 6, omitting non-person nodes and non-knows relationships.

Listing 7. Unstable results in Kùzu caused by an incorrect hash index scan, where the graph data can be found in the supplementary material.

```
MATCH (n) WHERE (n.p34 = 'aXKCUJeQQwHo') OR
(n.p45 = true) RETURN COUNT (*)
--{unstably returns 20 or 21 🐞}
```

shown in Listing 5, we detected a bug in nx.trophic_level() caused by the insufficient checking of whether all nodes can be reached by at least one basal node. For such graphs containing an unreachable node from basal nodes, nx.trophic_level() returned a random large number for the node, *e.g.,* 2.3804740887529764e+16. Graph-cutting can capture such insufficient checking using Lossless-Cutting, as each node's trophic level in the original graph should be equivalent to that in the subgraphs when there are no cutting edges. However, NetworkX will return two different large numbers for the same node when this insufficient checking occurs.

Incorrect pattern matching in Kùzu. Listing 6 shows a logic bug in Kùzu caused by an improper Worst-Case-Optimal Join when the query pattern contains cycles. In this case, we queried the number of mutual knowing relationships, *i.e.,* how many ordered pairs of people (a, b) such that a knows b and b knows a, in the tinysnb dataset as shown in Figure 7. However, Kùzu returned an incorrect result 16, while only 12 such pairs existed in the graph, *i.e.,* choosing two distinct people from $\{1, 2, 3, 4\}$. The root cause of this logic bug was that some relationships with types other than "knows" in the graph (omitted in the Figure 7), which disturbed the Worst-Case-Optimal Join for the query. Both Lossy-Cutting and Lossless-Cutting could detect the bug by separating those relationships.

Unstable predicate handling in Kùzu. Lisiting 7 shows another logic bug we detected in Kùzu. On the graph data provided in supplementary material, Kùzu failed to return results stably. Specifically, it returned 20 or 21 for a query aiming to count the nodes satisfying one of its properties equals to 'aXKCUJeQQwHo' and another equals to true. The root cause of this logic bug was related to Kùzu's incorrect implementation of the hash index scan. Since the results were unstable, the bug could be easily detected by Graph-cutting.

Table 4. Comparison of logic bug-inducing test cases and unique bugs in Kùzu across different approaches over 24 hours: 3 unique bugs detected by GSLICER were missed by all baseline strategies, and 3 baseline strategies identified three unique bugs that GSLICER overlooked.

Approach	No. bug-inducing cases	No. unique bugs
GraphGenie	952	1
GRev	4,428	3
GSLICER	253	4

5.2 Comparision with Baseline Strategies

GDBMS. We compare GSLICER in terms of bug-inducing cases and unique bugs found in Kùzu against two recent baseline strategies. These tools were selected, because they are the latest GDBMS testing methods that can be adapted to Kùzu with minimal source code modifications. Note that, we exclude earlier works [30, 33] and those focused on testing GDBMS using the Gremlin query language and its features [60–62], which is not supported by Kùzu.

(1) *GraphGenie* [25]: GraphGenie detects logic bugs in GDBMS by leveraging injective and surjective relations to generate comparable query pairs.
(2) *GRev* [36]: GRev equivalently rewrites GDBMSes queries by embedding them into an Abstract Syntax Graph and developing an algorithm to extract equivalent matching conditions from the structure.

Table 4 presents the results of testing Kùzu v0.4.1, highlighting the total number of logic bug-triggering cases and the unique bug counts among the random 100 reported instances. Due to the absence of a reliable bug deduplication tool and a checker for false alarms, we manually examined the results for each approach, similar to the methods used in previous works [25, 66]. Specifically, we manually filtered out false alarm cases by their features from the total number of bug-inducing test cases and identified duplicates to report the number of unique logic bugs.

Overall results. Over a 24-hour testing period, GraphGenie, GRev, and GSLICER generated 952, 4,428, and 253 valid logic bug reports, respectively. In the first 100 cases of each, we manually identified 1, 3, and 4 unique logic bugs in Kùzu v0.4.1, respectively. Although GSLICER produced fewer bug-inducing cases compared to the baselines, likely due to the additional time required by GRAPH-CUTTING for creating subgraphs, it discovered the highest number of unique logic bugs, demonstrating GRAPH-CUTTING's effectiveness. A deeper investigation revealed that GSLICER detected 3 unique logic bugs missed by both baseline strategies. Conversely, the baselines identified 3 logic bugs that GSLICER missed. This suggests that GSLICER not only supports testing general graph-processing systems, but also complements existing GDBMS testing approaches.

False alarms introduced by query mutation. During the evaluation, we identified a false alarm issue with existing GDBMS testing tools. This is because they rely on query mutation, and human-defined mutation rules are prone to errors when adapting to different target systems or when the syntax of query languages changes. For example, in the testing process of GraphGenie, the tool originally reported 31,880 bug-inducing cases. However, most stemmed from one false alarm case shown in Listing 8. Specifically, the two queries in Listing 8 are expected to return equivalent queries in GraphGenie by its mutation rule `CountOtherName` [25]. However, both queries returned the correct results in Kùzu, as null should be included in the count for the second query but not in the first one, which was also confirmed by the developers of Kùzu.

In contrast, GSLICER tests GDBMS by executing the same query on different graph data without relying on any hard-coded mutation rules. This approach significantly reduces the likelihood of

Listing 8. An false alarm case in Kùzu reported by GraphGenie.

```
1 OPTIONAL MATCH (n) WHERE false RETURN COUNT(n) --{0} ✓
2 OPTIONAL MATCH (n) WHERE false RETURN COUNT(*) --{1} ✓
```

Listing 9. A logic bug in Kùzu detected by GRev but missed by Gslicer, where the graph data is omitted.

```
1 MATCH (n1)-[]->(n2:L1:L7) RETURN COUNT(*) --{19} ✓
2 MATCH (n1)-[]->(n2:L1:L7), (n2) RETURN COUNT(*)
3 --{150} 🐞
```

false alarms. Consequently, no false alarms were found in the first 100 cases manually investigated from Gslicer.

Bugs missed by baselines. The logic bugs detected by Gslicer, while missed by baseline strategies, are mostly caused by *data interference*, *i.e.,* the query results are mistakenly influenced by data that should not contribute to the result. For example, the bug shown in Listing 6 is triggered only when non-person nodes and non-know relationships exist in the tinysnb dataset. If we keep only the graph data shown in Figure 7 in the dataset, the query returns the correct result 12. Therefore, regardless of how the query is mutated, the original and mutated queries are unlikely to yield results that violate the relationships established by baseline strategies, unless the unrelated data is removed. However, Graph-cutting, which naturally separates graph data, is more effective in detecting those bugs caused by data interference.

In addition, the bug shown in Listing 3 illustrates another type of bug that Gslicer excels at detecting, outperforming existing methods. The query in Listing 3 has been heavily optimized in Neo4j [41], allowing it to be executed in just a few seconds, despite having a search space of over 2^{63} subgraphs. However, the mutated queries are challenging to optimize, causing the GDBMS to fail in returning results within a practical time, *e.g.,* adding a `WHERE` clause before the `RETURN`. Therefore, we believe that Graph-cutting is also more effective in detecting bugs triggered by queries that are difficult to mutate through baseline strategies.

Bugs missed by Gslicer. Several logic bugs were missed by Gslicer during our evaluation. The queries that trigger these bugs exhibit consistent behavior regardless of the graph data, allowing them to always pass Graph-cutting. For example, Listing 9 shows a logic bug in Kùzu detected by GRev but missed by our approach. In this case, the second query is mutated by adding a redundant node `(n2)` to the pattern of the first query, which should not change the result. However, the second query returns an incorrect large result, which reveals a logic bug in Kùzu.second. Upon investigation, we found that the incorrect results caused by adding redundant nodes occur in Kùzu for any base query and are independent of the graph data, making it difficult for approaches other than query mutation, such as Gslicer, to detect the bug.

In conclusion, our experiments in Kùzu v0.4.1 demonstrate that Graph-cutting can serve as a complementary tool for testing GDBMS, and we believe it will uncover additional logic bugs overlooked by existing approaches in other versions of GDBMSes. It is less prone to false alarms and more effective at detecting logic bugs related to data interference or those triggered by queries that are difficult to mutate.

Graph library. To the best of our knowledge, Gslicer is the first to automatically detect logic bugs in graph libraries. Therefore, we compare Graph-cutting with differential testing [39] across equivalent APIs in NetworkX [20] and Neo4j-GDS [41], where multiple algorithms are available for some tasks. For example, four different algorithms are provided for computing shortest paths.[11] Based on this, we manually identify the available equivalent APIs and perform differential testing

[11] https://networkx.org/documentation/stable/reference/algorithms/shortest_paths

Table 5. Code coverage of Kùzu tested over 24 hours.

Approach	Line	Function
GSLICER	34.8%	20.1%
Unit Tests	89.3%	70.0%
Unit Tests + GSLICER	89.4%	70.3%

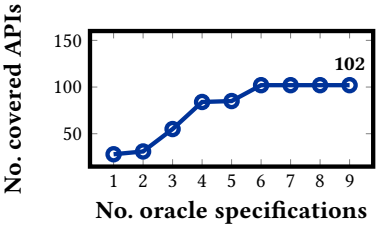


Fig. 8. The relationship between the number of GRAPH-CUTTING oracles and the number of APIs in NetworkX covered by them.

across them for the corresponding tasks associated with each logic bug we detected. Note that, it is impossible to implement a unified differential testing approach for all tasks in the graph library due to the lack of an accurate and efficient tool to automatically identify equivalent APIs.

As a result, we found one logic bug in NetworkX can be detected by differential testing between eigenvector_centrality_numpy and and eigenvector_centrality. For the remaining three logic bugs, differential testing could not identify them due to the lack of equivalent APIs, *e.g.,* the bug shown in Listing 4, where NetworkX provides only one API for the task, or because the root cause affects all equivalent APIs *e.g.,* both triangle counting and listing in Neo4j are impacted by the bug illustrated in Figure 1.

In conclusion, at least 8 logic bugs were detected by GSLICER that baseline strategies have missed.

5.3 Coverage

Code coverage. We evaluate the generators, *i.e.,* the graph and query generators of GSLICER, by collecting code coverage data from Kùzu, which offers insights into the components tested. For 24 hours, GSLICER was run on Kùzu, and the resulting code coverage was compared with the data from unit tests. Table 5 shows the results, including line and function coverage. GSLICER achieved approximately 35% line coverage and 20% function coverage, which, while relatively low, was expected. This is because GSLICER focuses primarily on graph data processing and query handling, whereas Kùzu contains other components unrelated to graph processing. For instance, around 29% of Kùzu's lines pertain to I/O functions for various formats and client APIs, which are less likely to be exercised by graph generation and query execution. In the core graph processing functions, GSLICER achieved higher coverage. For example, it triggers all lines in the "*hash_join*" and "*intersect*" functions, both of which contain a logic bug detected by GSLICER.

Additionally, GSLICER covered 63 lines and 36 functions that were missed by the unit tests, highlighting its ability to provide supplementary coverage. After manually investigating all logic bugs found by GSLICER in Kùzu, we confirmed that none of the bugs triggered new lines uncovered by the unit tests. This suggests that GSLICER can detect new bugs even in code that unit tests have thoroughly tested.

Task coverage. We applied the mechanism outlined in Algorithm 1 to 366 graph-processing APIs in NetworkX, using 9 basic GRAPH-CUTTING oracle specifications with $\tau = 20$. These specifications treat all adjustment functions as identity functions $\mathcal{I}$, and define $\{(\oplus, \prec)\} = \{\cup, +, \max/\min\} \times \{=, \geq, \leq\}$. The number of APIs covered by these GRAPH-CUTTING oracles offers insights into GRAPH-CUTTING's generalizability and the effectiveness of the automated oracle specification mechanism, as well as helps us identify which graph tasks can and cannot be tested by GRAPH-CUTTING.

The relationship between the number of specifications and the number of APIs that passed the mechanism is illustrated in Figure 8. Our mechanism can cover 102 APIs (~ 28%) by using only a few of Graph-cutting oracles, highlighting the generalizability of Graph-cutting. As expected, no additional improvement is observed beyond 7 oracles, since many graph tasks are covered by multiple oracles. For instance, is_regular checks whether every vertex in a graph G has the same degree, returning $\texttt{res}(G)$ from $\{0, 1\}$. In Lossless-cutting, both $\texttt{res}(G^{(1)}) + \texttt{res}(G^{(2)}) \leq \texttt{res}(G)$ and $\max\{\texttt{res}(G^{(1)}), \texttt{res}(G^{(2)})\} \geq \texttt{res}(G)$ hold, as each connected component being regular is a necessary condition for the entire graph to be regular. Additionally, these are just the simplest oracles based on Graph-cutting. We believe that by applying more specifications, more tasks within graph libraries can be effectively tested, *e.g.,* page rank needs a complex Graph-cutting oracle shown in Table 1.

We further sampled 10 APIs from which we automatically detected oracles and manually verified their correctness. As a result, one of the authors identified 8 of the 10 APIs sampled as correct Graph-cutting oracles for their corresponding graph tasks in a total of 15 minutes. The remaining 2 oracles are incorrect due to an insufficient number of small-scale graph data, which made it difficult to find a counter-example, *i.e.,* τ is too small. However, this is expected because increasing τ also makes the identified oracles less likely to capture logic bugs in their corresponding graph tasks, because logic bugs can also let the API fail to pass a correct oracle in our mechanism. By adjusting τ, Gslicer offers flexibility for users to balance correctness and effectiveness when using the automatically identified oracles. Note that, these false oracles will not trigger false alarms in Gslicer, as we manually verify them before applying them to test the graph library, and this manual checking requires significantly less effort than identifying oracles.

5.4 Efficiency Analysis

Runtime distribution. In this experiment, we evaluate the runtime distribution across three components of Gslicer and the target system, varying the size of the generated original graph *i.e.,* $N + M$, the sum of vertices and edges. For each run, we randomly generated three input graphs and executed 100 random queries on these graphs using Kùzu, reporting the average runtime distribution over 10 runs as shown in Figure 9. The results show that the execution time on the target systems dominates the overall run time (~99%). Our experiment on GraphGenie [25] also confirms this observation, where the execution time spent on Kùzu accounts for ~89% of the total runtime. While we do not claim that our oracle runs faster than state-of-the-art methods, as run-time distribution depends on the complexity of generated queries and thus differences in query generators, these findings highlight that database execution is the primary bottleneck. Furthermore, we also find that, as the graph size increases, the run time for Gslicer's graph generator and Kùzu's execution scales proportionally, while the time spent in the query generator and validator remains nearly constant.

Oracle design cost. In addition to the manual effort involved in checking oracles from Algorithm 1, we evaluate the cost of manually designing Graph-cutting oracles for various graph algorithms in NetworkX. Specifically, we randomly sampled 10 algorithms from NetworkX, and one of the authors attempted to design both Lossy-cutting and Lossless-cutting oracles for each algorithm. Within an overall 25 minutes, the author successfully designed at least one valid Graph-cutting oracle for 7 out of the 10 algorithms, comprising 7 Lossless-cutting oracles and 3 Lossy-cutting oracles. For the remaining three algorithms, the failure to design Graph-cutting oracles was mainly due to the algorithms' inputs not being graphs, *e.g.,* a degree sequence. We believe that, compared to designing traditional tests, creating Graph-cutting oracles reduces manual effort and facilitates automated subsequent testing.

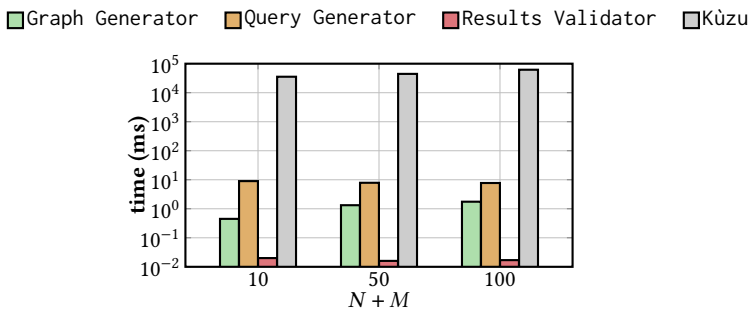


Fig. 9. Gslicer's run-time distribution on Kùzu.

Listing 10. An illustrative example of how Graph-cutting tests Gremlin-based GDBMSes.

```
g.addV('vertex').as('v1');
g.addV('vertex').as('v2');
g.addV('vertex').as('v3');
g.addV('vertex').as('v4');
g.addE('edge').from('v1').to('v2');
g.addE('edge').from('v3').to('v4');
g.V().as('a').out().as('b').count(); --{2}
g.V(1, 2).as('a').out().as('b').count(); --{1}
g.V(3, 4).as('a').out().as('b').count(); --{1}
// test oracle: line 6 = line 7 + line 8
```

6 Discussion

Effectiveness. In detecting unique and previously unknown bugs, Gslicer has several advantages over existing approaches: (1) it can test a more general range of graph-processing systems. To the best of our knowledge, Gslicer is the first automated tool designed for testing graph libraries, successfully detecting 5 unique and previously unknown logic bugs; (2) for traditional GDBMS testing, Gslicer can identify logic bugs related to data interference as well as those triggered by queries that are difficult to mutate. These types of logic bugs are often overlooked by existing query-mutation-based methods. Additionally, since it does not rely on hard-coded rules for query transformation, Gslicer is less likely to generate false-positive reports.

Testing other kinds of GDBMSes. Graph-cutting is a general approach that can be applied to other types of GDBMSes using query languages such as Gremlin, as its correctness is guaranteed for all GPM tasks. Testing such systems with Graph-cutting could be implemented only with additional engineering effort. For example, to test Gremlin-based GDBMSes, once the traversal of a Gremlin query is determined and connected, the counting results can be verified like for Cypher queries. For instance, as shown in Listing 10, the result of the Gremlin query for counting edges in the original graph should be equal to the sum of the edges in the two disconnected subgraphs. Alternatively, one could leverage the official Gremlin server plugin, Cypher for Gremlin, to automatically translate Cypher queries into Gremlin queries, similar to a previous method [25].

Applicability. For testing GDBMS, we directly use the Graph-cutting oracles for GPM and successfully detected several logic bugs that have been overlooked by existing approaches. For graph library, our evaluation demonstrates that Graph-cutting is versatile and applicable to numerous graph tasks. We believe this versatility arises from the common observation that certain graph patterns are critical across various tasks in graph-processing systems. However, some functionalities

related to randomization in graph-processing systems cannot effectively be tested by Gslicer. For instance, when GDBMS queries contain randomization clauses, *e.g.,* `rand()`, no Graph-cutting oracles will apply due to ambiguous results. Additionally, some graph tasks in graph libraries involve randomization, such as node embedding based on random walks [19]. Another limitation is that our testing does not apply to transactions and data manipulations in GDBMSes, and cannot cover all graph tasks in graph libraries.

Potential manual efforts. First, while Graph-cutting can be applied to various graph tasks and directly test GDBMSes, it sometimes requires different oracle specifications tailored to specific APIs in graph libraries, which needs additional engineering efforts as shown in Section 5.4. However, existing work in GDBMS testing also needs similar efforts to identify numerous rules for query mutations [25, 66]. To reduce the engineering effort, we designed an automated mechanism in Gslicer to identify those concrete Graph-cutting oracles, for which users only need to check their correctness. Second, in our evaluation, we manually deduplicated logic bugs. However, deduplication is not necessary if the developers fix the bug after we report it, because we can resume testing the new version. Test deduplication is an open challenge [15] in the field. As such, manual deduplication remains a standard practice across the field, including in our approach.

Threats to validity. First, due to the lack of effective bug deduplication tools in graph-processing systems and the large number of reports generated, we must sample to manually deduplicate and identify unique bugs, which may result in missing some bugs. To ensure the evaluation is as accurate as possible, we carefully analyze both the bug patterns and root causes in all sampled reports. We also confirm the false alarm cases in the baseline strategies with the developers. Second, we do not cover all GDBMSes and graph libraries. To mitigate the threat, we selected three open-source graph-processing systems with active development communities.

7 Related Work

Testing GDBMSes. Various works have focused on the reliability of GDBMSs. As discussed throughout this paper, existing approaches focus on detecting logic bugs and are based on either differential testing or query mutation. For instance, GDsmith [33] utilizes differential testing to detect logic bugs in Cypher-based GDBMSs, while Grand [62] targets Gremlin-based GDBMSs. However, due to the inherent limitations of differential testing in handling intended inconsistencies across different GDBMS versions, query-mutation-based approaches have been proposed. These methods generate and execute related queries within the same GDBMS. GDBMeter [30] employs the TLP oracle [46], GraphGenie [25] explores injective and surjective relationships, GRev [36] uses abstract syntax graphs to embed patterns for equivalent query generation, and Gamera constructs graph-aware test oracles to mutate queries. Additionally, GraspDB [34] focuses on detecting logic bugs in GDBMSes' write operations, while several approaches specifically designed for testing Gremlin-based GDBMSs have also been proposed [60, 61]. In this paper, we propose Graph-cutting, a general approach that can be applied to a broader range of graph-processing systems but is also effective in testing GDBMSes. This approach is novel in that it is neither based on differential testing nor query mutation, allowing it to effectively detect logic bugs in GDBMSes related to data interference or queries that are challenging to mutate.

Testing DBMSes. In addition to testing GDBMSes, automated testing approaches have been developed to detect various types of bugs in relational DBMSes. Most methods for testing relational DBMSes focus on security-related bugs, such as memory errors, which do not require an explicit test oracle. Several works have been proposed based on grey-box fuzzing and efficient test case

generation [26, 63]. Detecting logic bugs is challenging due to the necessity of test oracles to verify query result correctness. Several tools [4, 5, 28, 45–47, 52, 59, 64] identify logic bugs in SQL handling by examining relationships between related query results. DQE [49] focuses on `UPDATE` and `DELETE` queries, ensuring consistent row access via the same predicate, while TxCheck [27] detects transaction logic bugs using random topological sorting of acyclic statement-dependency graphs. Research on detecting performance issues in relational DBMSs includes APOLLO [29], which compares execution times across different DBMS versions, AMOEBA [35], which assesses semantically equivalent query pairs, and CERT [3], whose test oracle inspects cardinality estimates. In addition to testing relational DBMSs, Deng et al. proposed Spatter, an automated testing tool for Spatial databases [14].

Metamorphic testing. Metamorphic testing [12] is one of the most successful methods to detect logic bugs. It generates two related test inputs and validates the consistency of their results. Metamorphic testing uses an input I to a system and its corresponding output O to derive a new input I' (with output O'). A test oracle can then be established to verify whether a so-called *Metamorphic Relation* holds between O and O'. Graph-cutting can be regarded as a metamorphic testing-based approach if we also take the graph data as graph-processing systems' inputs. Other than data management systems, it has successfully improved the reliability of various systems, such as compilers [31, 51], machine learning systems [10, 11, 21, 57, 58], and Datalog engines [37].

8 Conclusion

This paper has presented Graph-cutting, a novel universal approach for detecting logic bugs in graph-processing systems. By establishing relationships between query results in the original graph and subgraphs, our method has effectively detected several bugs in the systems. The implementation of Gslicer has demonstrated its capability by uncovering 39 unique and previously unknown bugs, with 34 being fixed by developers, highlighting its practical utility. Moreover, our evaluations have revealed that Gslicer successfully identified logic bugs that baseline strategies overlooked. We believe that the techniques developed in this work have significant potential for enhancing the reliability and quality of graph-processing systems across a wide range of applications.

9 Acknowledgements

We thank the anonymous SIGMOD reviewers for their valuable feedback on the earlier draft of this paper. We also want to thank all the graph-processing systems developers for responding to our bug reports as well as analyzing the bugs we reported. This research was supported by a Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant as well as the National Research Foundation, Singapore, Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing), the Guangdong Basic and Applied Basic Research Foundation (No. 2024A1515010145), and the Shenzhen Science and Technology Program (Shenzhen Key Laboratory Grant No. ZDSYS20230626091302006). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore.

References

[1] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[2] Jinsheng Ba and Manuel Rigger. Testing database engines via query plan guidance. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2060–2071. IEEE, 2023.

[3] Jinsheng Ba and Manuel Rigger. 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*, pages 1–13, 2024.

[4] Jinsheng Ba and Manuel Rigger. An exploratory case study of query plan representations, 2024.

[5] Jinsheng Ba and Manuel Rigger. Keep it simple: Testing databases via differential query plans. *Proceedings of the ACM on Management of Data*, 2(3):1–26, 2024.

[6] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[7] Stephen P Borgatti. Structural holes: Unpacking burt's redundancy measures. *Connections*, 20(1):35–38, 1997.

[8] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. Motif counting beyond five nodes. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 12(4):1–25, 2018.

[9] Da Cao, Liqiang Nie, Xiangnan He, Xiaochi Wei, Shunzhi Zhu, and Tat-Seng Chua. Embedding factorization models for jointly recommending items and user generated lists. In *Proceedings of the 40th international ACM SIGIR conference on research and development in information retrieval*, pages 585–594, 2017.

[10] Jialun Cao, Meiziniu Li, Yeting Li, Ming Wen, Shing-Chi Cheung, and Haiming Chen. Semmt: a semantic-based testing approach for machine translation systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–36, 2022.

[11] Jialun Cao, Yaojie Lu, Ming Wen, and Shing-Chi Cheung. Testing coreference resolution systems without labeled test sets. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 107–119, 2023.

[12] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543*, 2020.

[13] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. Differentially testing database transactions for fun and profit. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[14] Wenjing Deng, Qiuyang Mang, Chengyu Zhang, and Manuel Rigger. Finding logic bugs in spatial database engines via affine equivalent inputs. *Proceedings of the ACM on Management of Data*, 2(6):1–26, 2024.

[15] Alastair F Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1017–1032, 2021.

[16] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. Kùzu graph database management system. In *CIDR*, 2023.

[17] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*, pages 1433–1445, 2018.

[18] Brian Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*, volume 45, 2006.

[19] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[20] Aric Hagberg, Pieter J Swart, and Daniel A Schult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), 2008.

[21] Pinjia He, Clara Meister, and Zhendong Su. Testing machine translation via referential transparency. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 410–422. IEEE, 2021.

[22] Lin Hu, Lei Zou, and Yu Liu. Accelerating triangle counting on gpu. In *Proceedings of the 2021 International Conference on Management of Data*, pages 736–748, 2021.

[23] Kai Huang, Haibo Hu, Qingqing Ye, Kai Tian, Bolong Zheng, and Xiaofang Zhou. Ted: Towards discovering top-k edge-diversified patterns in a graph database. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023.

[24] ISO/IEC JTC1. Database Language GQL – 2024 Article 39075. https://jtc1info.org/wp-content/uploads/2024/04/2024-Article-39075-Database-Language-GQL.docx.pdf, 2024. [Online; accessed 11-Oct-2024].

[25] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland HC Yap, Zhenkai Liang, and Manuel Rigger. Detecting logic bugs in graph database management systems via injective and surjective graph query transformation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[26] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. {DynSQL}: Stateful fuzzing for database management systems with complex and valid {SQL} query generation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4949–4965, 2023.

[27] Zu-Ming Jiang and Zhendong Su. Detecting logic bugs in database engines via equivalent expression transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 821–835, 2024.

[28] Zu-Ming Jiang and Zhendong Su. Detecting logic bugs in database engines via equivalent expression transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 821–835, 2024.

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

[30] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. Testing graph database engines via query partitioning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 140–149, 2023.

[31] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 216–226, New York, NY, USA, 2014. Association for Computing Machinery.

[32] Stephen Levine. Several measures of trophic structure applicable to complex food webs. *Journal of theoretical Biology*, 83(2):195–207, 1980.

[33] Wei Lin, Ziyue Hua, Luyao Ren, Zongyang Li, Lu Zhang, and Tao Xie. Gdsmith: Detecting bugs in graph database engines. *arXiv preprint arXiv:2206.08530*, 2022.

[34] Shuang Liu, Junhao Lan, Xiaoning Du, Jiyuan Li, Wei Lu, Jiajun Jiang, and Xiaoyong Du. Testing graph database systems with graph-state persistence oracle. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 666–677, 2024.

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

[36] Qiuyang Mang, Aoyang Fang, Boxi Yu, Hanfei Chen, and Pinjia He. Testing graph database systems via equivalent query rewriting. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[37] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. Metamorphic testing of datalog engines. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 639–650, 2021.

[38] Wim Martens, Matthias Niewerth, Tina Popp, Stijn Vansummeren, and Domagoj Vrgoc. Representing paths in graph database pattern matching. *arXiv preprint arXiv:2207.13541*, 2022.

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

[40] Atsushi Miyauchi, Tianyi Chen, Konstantinos Sotiropoulos, and Charalampos E Tsourakakis. Densest diverse subgraphs: How to plan a successful cocktail party with diversity. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 1710–1721, 2023.

[41] Neo4j. Neo4j, 2024.

[42] Lawrence Page. The pagerank citation ranking: Bringing order to the web. Technical report, Technical Report, 1999.

[43] Aduri Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Counting and sampling triangles from a graph stream. *Proceedings of the VLDB Endowment*, 6(14):1870–1881, 2013.

[44] Pedro Ribeiro, Pedro Paredes, Miguel EP Silva, David Aparicio, and Fernando Silva. A survey on subgraph counting: concepts, algorithms, and applications to network motifs and graphlets. *ACM Computing Surveys (CSUR)*, 54(2):1–36, 2021.

[45] Manuel Rigger and Zhendong Su. 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*, pages 1140–1152, 2020.

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

[47] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 667–682, 2020.

[48] Jari Saramäki, Mikko Kivelä, Jukka-Pekka Onnela, Kimmo Kaski, and Janos Kertesz. Generalizations of the clustering coefficient to weighted complex networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, 75(2):027105, 2007.

[49] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. Testing database systems via differential query execution. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2072–2084. IEEE, 2023.

[50] Mauro Sozio and Aristides Gionis. The community-search problem and how to plan a successful cocktail party. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 939–948, 2010.
[51] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*, pages 849–863, 2016.
[52] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. Detecting logic bugs of join optimizations in dbms. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023.
[53] Kaixin Wang, Kaiqiang Yu, and Cheng Long. Efficient k-clique listing: An edge-oriented branching strategy. *Proceedings of the ACM on Management of Data*, 2(1):1–26, 2024.
[54] Pinghui Wang, Yiyan Qi, Yu Sun, Xiangliang Zhang, Jing Tao, and Xiaohong Guan. Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage. *Proceedings of the VLDB Endowment*, 11(2):162–175, 2017.
[55] Daniel ten Wolde, Gábor Szárnyas, and Peter Boncz. Duckpgq: Bringing sql/pgq´ to duckdb. *Proceedings of the VLDB Endowment*, 16(12):4034–4037, 2023.
[56] Dominic Wüst, Zu-Ming Jiang, and Zhendong Su. Dinkel: Testing graph database engines via state-aware query generation. *arXiv preprint arXiv:2408.07525*, 2024.
[57] Boxi Yu, Yiyan Hu, Qiuyang Mang, Wenhan Hu, and Pinjia He. 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*, pages 883–894, 2023.
[58] Boxi Yu, Zhiqing Zhong, Xinran Qin, Jiayi Yao, Yuancheng Wang, and Pinjia He. Automated testing of image captioning systems. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 467–479, 2022.
[59] Chi Zhang, Linzhang Wang, and Manuel Rigger. Finding cross-rule optimization bugs in datalog engines. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):110–136, 2024.
[60] Yingying Zheng, Wensheng Dou, Lei Tang, Ziyu Cui, Yu Gao, Jiansen Song, Liang Xu, Jiaxin Zhu, Wei Wang, Jun Wei, et al. Testing gremlin-based graph database systems via query disassembling. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1695–1707, 2024.
[61] Yingying Zheng, Wensheng Dou, Lei Tang, Ziyu Cui, Jiansen Song, Ziyue Cheng, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. Differential optimization testing of gremlin-based graph database systems. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 25–36. IEEE, 2024.
[62] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. 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*, pages 302–313, 2022.
[63] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. Squirrel: Testing database management systems with language validity and coverage feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 955–970, 2020.
[64] Suyang Zhong and Manuel Rigger. Scaling automated database system testing, 2025.
[65] Yingli Zhou, Qingshuo Guo, Yixiang Fang, and Chenhao Ma. A counting-based approach for efficient k-clique densest subgraph discovery. *Proceedings of the ACM on Management of Data*, 2(3):1–27, 2024.
[66] Zeyang Zhuang, Penghui Li, Pingchuan Ma, Wei Meng, and Shuai Wang. Testing graph database systems via graph-aware metamorphic relations. *Proceedings of the VLDB Endowment*, 17(4):836–848, 2023.
[67] Lei Zou, Lei Chen, and M Tamer Özsu. Distance-join: Pattern match query in a large graph database. *Proceedings of the VLDB Endowment*, 2(1):886–897, 2009.

Received October 2024; revised January 2025; accepted February 2025