Testing Gremlin-Based Graph Database Systems via Query Disassembling

Yingying Zheng[*]
Wensheng Dou[*†‡]
Institute of Software at CAS, China

Lei Tang[*]
Ziyu Cui[*]
Institute of Software at CAS, China

Yu Gao[*]
Jiansen Song[*]
Institute of Software at CAS, China

Liang Xu
Jinling Institute of Technology, China

Jiaxin Zhu[*†]
Institute of Software at CAS, China

Wei Wang[*†]
Institute of Software at CAS, China

Jun Wei[*†]
Institute of Software at CAS, China

Hua Zhong[*]
Institute of Software at CAS, China

Tao Huang[*]
Institute of Software at CAS, China

Abstract

Graph Database Systems (GDBs) support efficiently storing and retrieving graph data, and have become a critical component in many important applications. Many widely-used GDBs utilize the Gremlin query language to create, modify, and retrieve data in graph databases, in which developers can assemble a sequence of Gremlin APIs to perform a complex query. However, incorrect implementations and optimizations of GDBs can introduce logic bugs, which can cause Gremlin queries to return incorrect query results, e.g., omitting vertices in a graph database.

In this paper, we propose *Query Disassembling* (`QuDi`), an effective testing technique to automatically detect logic bugs in Gremlin-based GDBs. Given a Gremlin query Q, `QuDi` disassembles Q into a sequence of atomic graph traversals $TList$, which shares the equivalent execution semantics with Q. If the execution results of Q and $TList$ are different, a logic bug is revealed in the target GDB. We evaluate `QuDi` on six popular GDBs, and have found 25 logic bugs in these GDBs, 10 of which have been confirmed as previously-unknown bugs by GDB developers.

CCS Concepts

• **Information systems** → *Database query processing*; • **Software and its engineering** → *Software testing and debugging*.

Keywords

Graph database systems, graph traversal, logic bug, bug detection

ACM Reference Format:
Yingying Zheng, Wensheng Dou, Lei Tang, Ziyu Cui, Yu Gao, Jiansen Song, Liang Xu, Jiaxin Zhu, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2024. Testing Gremlin-Based Graph Database Systems via Query Disassembling. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680392

[*]Affiliated with Key Lab of System Software at CAS, State Key Lab of Computer Science at Institute of Software at CAS, and University of CAS, Beijing. CAS is the abbreviation of Chinese Academy of Sciences.
[†]Affiliated with Nanjing Institute of Software Technology, University of CAS, Nanjing.
[‡]Wensheng Dou (wsdou@otcaix.iscas.ac.cn) is the corresponding author.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680392

1 Introduction

Graph Database Systems (GDBs) [58] support efficient storage and queries for graph data, which consists of vertices and edges. The popularity of GDBs has increased dramatically recently, and GDBs have played a significant role in many important applications, e.g., social networks [35], knowledge graphs [26, 45], and fraud detection [50]. According to the latest DB-Engines Ranking of GDBs [3], there have already been 41 widely-used GDBs, e.g., Neo4j [12], OrientDB [16], NebulaGraph [15], JanusGraph [8], and TigerGraph [33].

Unlike relational database systems that utilize *declarative* Structured Query Language (SQL) to access *relational* data, e.g., MySQL [10], MariaDB [9], TiDB [21], and PostgreSQL [17], GDBs do not have a standardized way to access *graph* data, and usually utilize their own query languages, e.g., nGQL [11] in NebulaGraph and GSQL [32] in TigerGraph. However, about half of the GDBs in the DB-Engines Ranking [3], e.g., Neo4j [12], OrientDB [16], JanusGraph [8], HugeGraph [6], TinkerGraph [22], and ArcadeDB [14], support the *procedural* Gremlin query language [4, 54], which is developed by Apache TinkerPop [22]. We refer to these GDBs that support the Gremlin query language as Gremlin-based GDBs.

The Gremlin query language used in Gremlin-based GDBs has totally different syntaxes and query patterns with SQL in relational database systems. Specifically, the Gremlin query language provides a group of Gremlin APIs to create, modify, and retrieve graph data. Developers can further assemble a sequence of Gremlin APIs and generate complex queries to achieve complex graph analyses.

To improve the performance of Gremlin queries, GDBs usually adopt complex execution and optimization strategies [5, 7], e.g., reordering filtering operations to execute cheaper operations first and merging filtering conditions for efficient graph queries. Unsurprisingly, the above complexity poses a major correctness challenge for GDBs. Incorrect implementations and optimizations of GDBs can introduce logic bugs, which can silently cause the GDBs to return an incorrect query result for a given Gremlin query without crashing the GDBs, e.g., omitting vertices in a graph database.

Figure 1 shows a real-world logic bug ArcadeDB#500 that we detected in ArcadeDB [14]. In this logic bug, the Gremlin query (Line 1)

```
1 g.V().has('person','age',lt(30)).hasLabel('person','book');
2   -- v:{1,2,3,4} ✘
3   -- v:{1,4} ✔
```

Figure 1: A logic bug ArcadeDB#500 detected by our approach in ArcadeDB [14]. This Gremlin query wrongly retrieves all vertices, rather than persons who are less than 30 years old in Figure 2.

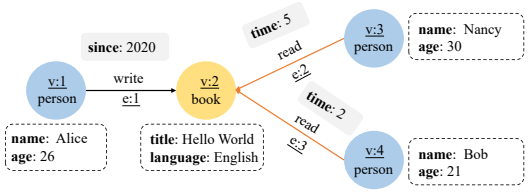

Figure 2: A labeled property graph.

in Figure 1 first retrieves all vertices of the graph shown in Figure 2 ($g.V()$), and then keeps *person* vertices in which the *age* property is less than 30 ($has('person','age',lt(30))$), and then keeps vertices that have a label *person* or *book* ($hasLabel('person','book')$). For this query, its correct result should be v:{1, 4}. However, ArcadeDB returns an incorrect result v:{1, 2, 3, 4}. ArcadeDB developers explained that this logic bug was caused by the incorrect implementation for the assembly of multiple filtering conditions, and fixed it quickly after we submitted it.

Logic bugs in GDBs are likely to go unnoticed by GDB developers, because we lack an effective test oracle for automatic testing to verify whether a GDB behaves correctly for a given Gremlin query, thus detecting logic bugs. Some approaches [39, 62, 64] adopt differential testing [49] to reveal discrepancies among multiple GDBs for the same graph database and graph queries. The clear drawback of this technique is that it cannot detect the same bugs that exist in all target GDBs and can only be used to test common features in target GDBs. GDBMeter [42] adopts Ternary Logic Partitioning (TLP) [52] to derive a graph query into three disjoint sub-queries based on a randomly generated predicate, and only focuses on detecting predicate-related bugs. Furthermore, some testing approaches [27, 30, 37, 51–53, 56, 57] have proposed new test oracles to effectively find logic bugs in an individual relational database system. However, these approaches cannot be applied on Gremlin-based GDBs because the *procedural* Gremlin query language adopts different syntaxes and query patterns from *declarative* SQL.

We observe that a complex Gremlin query can be disassembled into a sequence of atomic graph traversals (an atomic graph traversal can achieve one-step traversal in the graph database), which shares the equivalent execution semantics as the original query. Inspired by this observation, we propose *Query Disassembling* (QuDi), an effective test oracle to reveal logic bugs in individual Gremlin-based GDBs. Specifically, given a Gremlin query Q, we disassemble it into a sequence of atomic graph traversals $TList =< T_1, T_2, ..., T_n >$, in which T_i denotes the i-th atomic graph traversal in Q. Then, for an atomic graph traversal T_i in $TList$, we construct a query, which takes the result $RS_{T_{i-1}}$ of its previous traversal T_{i-1} as input, and computes its result RS_{T_i}. In such case, query Q and its disassembled graph traversal sequence $TList$ share the equivalent execution semantics, and should obtain the same result. That said, query Q's result RS_Q must be equal to the result RS_{T_n} of the last traversal T_n in $TList$. Otherwise, a potential logic bug is detected in the target GDB. We further propose three execution strategies to implement this test oracle, aiming at finding more logic bugs. By disassembling a complex Gremlin query into a sequence of atomic graph traversals, QuDi can prevent some GDB optimizations from kicking in, and thus can detect logic bugs caused by the assembly of atomic graph traversals and related optimizations.

We implement the above technique as QuDi, and evaluate QuDi on six popular Gremlin-based GDBs, i.e., Neo4j [12], OrientDB [16], JanusGraph [8], HugeGraph [6], TinkerGraph [22], and ArcadeDB [14]. At the time of writing this paper, we have found 25 logic bugs in these GDBs. Among them, 17 logic bugs have been confirmed by GDB developers, in which, 10 bugs are classified as previously-unknown bugs, and 7 bugs are classified as duplicate to existing bugs. 8 bugs have been fixed by GDB developers. For the 17 confirmed bugs, 11 bugs are caused by incorrect implementations and optimizations about the assembly of atomic graph traversals, and the remaining 6 bugs are caused by incorrect implementations of atomic graph traversals.

The idea of *query disassembling* is inspired by the following key observation: A complex query Q for a database system can be disassembled into multiple atomic queries *AtomicQ*s, and the assembly of *AtomicQ*s can compute the same query result as Q. In this paper, we only apply the idea of *query disassembling* on Gremlin-based GDBs. However, query disassembling is a general idea, and can be potentially applicable on other database systems. First, some other graph query languages e.g., Cypher [24] and SPARQL [18], also support the above key observation, and we can extend query disassembling on those GDBs that support these query languages. Second, SQL queries in relational database systems can support sub-queries and joins (e.g., SELECT * FROM (SELECT * FROM t1 JOIN t2)), which can also be disassembled into atomic queries. More discussions can be found in Section 5.

In summary, we make the following contributions in this paper.

- We propose a general and effective test oracle, query disassembling, for finding logic bugs in individual GDBs. By disassembling a complex query into an equivalent atomic graph traversal sequence, we can reveal logic bugs related to incorrect implementations and optimizations of Gremlin queries in GDBs.
- We evaluate QuDi on six widely-used GDBs. In total, we have detected 25 logic bugs in them, 10 of which have been confirmed as previously-unknown bugs by GDB developers.

2 Preliminaries

We first explain the labeled property graph model adopted by many GDBs (Section 2.1), and then introduce the Gremlin query language (Section 2.2) and its traversal model (Section 2.3). Finally, we briefly explain the Gremlin queries' execution (Section 2.4).

2.1 Labeled Property Graph Model

Most GDBs are built on the *labeled property graph model* [58] to efficiently store and retrieve graph data. A labeled property graph model consists of a set of vertices and edges associated with these vertices. Each vertex (edge) has a label to divide it into different

```
Vertex ::= g.V() || Vertex.[Filter | out() | in() | both() | order().by()] || Edge.[outV() | inV() | bothV()]
  Edge ::= g.E() || Edge.[Filter | order().by()] || Vertex.[outE() | inE() | bothE()]
Filter ::= has(Predicate) || hasNot() || hasLabel() || where(Predicate) || sample() || [not | and | or]([Filter |Vertex |Edge])
Predicate ::= eq() || neq() || lt() || lte() || gt() || gte() || inside() || outside() || between() || [not | and | or](Predicate)
Value ::= [Vertex | Edge].count() || [Vertex | Edge].values().[sum() | mean() | min() | max() | count()]
```

Figure 3: The abstract traversal model in the Gremlin query language [64].

vertex (edge) groups, and has a set of properties to describe its attributes. Figure 2 shows a labeled property graph that consists of four vertices and three edges. Specifically, three vertices with label *person* (i.e., v:1, v:3 and v:4) have two properties, i.e., *name* and *age*, while a vertex with label *book* (i.e., v:2) has *title* and *language* properties whose values are 'Hello World' and 'English', respectively. One edge labeled by *write* (i.e., e:1) has a *since* property with value '2020', while two edges with label *read* (i.e., e:2 and e:3) have a *time* property with value 5 and 2, respectively. All three edges associate *person* vertices to *book* vertices, and they are directed. For example, edge e:1 means that *person* v:1 writes *book* v:2.

2.2 Gremlin Query Language

GDBs utilize graph query languages to create, modify and retrieve graph data in graph databases. Many graph query languages have been proposed for GDBs [25]. For example, Apache TinkerPop [23] develops Gremlin [4, 54], Neo4j [38] develops Cypher [36], NebulaGraph [15, 59] develops nGQL [11], and TigerGraph [33] develops GSQL [32], to retrieve graph data from graph databases. According to the DB-Engines Ranking of GDBs [3], Gremlin is one of the most popular and widely-used graph query language, which has been supported by about half of the 41 GDBs. Specially, 6 out of the top 10 GDBs support Gremlin APIs.

Gremlin is a functional and procedural query language, and allows developers to assemble a sequence of Gremlin APIs to form complex queries. Specifically, a Gremlin query, starting with a Gremlin traversal source g, can traverse a labeled property graph by assembling a sequence of Gremlin APIs after g. For example, the Gremlin query in Figure 1 consists of four Gremlin APIs, i.e., $V()$, $has()$, $lt()$ and $hasLabel()$, in which $V()$ retrieves all vertices from the graph data in Figure 2, $has()$ and $hasLabel()$ keep vertices that satisfy the given filtering conditions, and $lt()$ is a parameter in $has()$. Besides these Gremlin query APIs, Gremlin also provides a series of update APIs. For example, we can add vertices (edges) by $addV()$ ($addE()$) in a graph database, and delete vertices or edges by $drop()$. In this paper, we mainly focus on Gremlin query APIs.

2.3 Gremlin Traversal Model

A Gremlin query consists of a sequence of Gremlin APIs, which are linked together. Zheng et al. [64] propose an abstract Gremlin traversal model to explain how to construct valid Gremlin queries, as shown in Figure 3. A key insight behind this traversal model is that the input type of a Gremlin API in a query should match the output type of its previous Gremlin API.

In this traversal model, `Vertex` describes the operations that return a vertex list, e.g., Gremlin API $V()$ that retrieves all vertices from a graph database can return a vertex list. `Edge` represents the operations that return a list of edges, e.g., we can use $g.V().outE()$, to retrieve the outgoing edges of all vertices in a graph database. `Filter` is a group of filtering operations, e.g., $has()$ and $hasLabel()$, which map entities that satisfy the given filtering conditions. The operations belonging to `Filter` can be assembled after `Vertex` and `Edge`, and return a list of vertices and edges, respectively. For example, if we assemble $g.V()$ and $has()$ together, the assembling query $g.V().has()$ also returns a vertex list. `Predicate` contains a group of predicate operations, e.g., $lt()$ that is used as a parameter of `Filter`. `Value` describes the operations that return a value or a value list, e.g., we can retrieve properties of vertices by $g.V().values()$.

Note that in this abstract traversal model, we ignore concrete parameters for each Gremlin API, e.g., *person* in the Gremlin API $hasLabel('person')$, and ignore some extra constraints, e.g., $sum()$ can only be used with a Number type.

2.4 Gremlin Query Execution

Gremlin queries are processed by *Gremlin Traversal Machine* (GTM) [54]. GTM adopts five categories of strategies [54] at Gremlin query compiling and execution, i.e., decoration, optimization, vendor optimization, finalization and verification. In these strategies, optimization strategies [4] aim to determine the most optimal execution plan according to the costs of accessing graph data. For example, optimization strategies can reorder filtering operations to execute cheaper filters first (i.e., `FilterRankingStrategy`) or merge operations for efficient searches (i.e., `IncidentToAdjacentStrategy`). Besides, Gremlin allows GDB developers to develop their own optimization strategies for efficiently executing Gremlin queries [5]. For example, `TinkerGraphCountStrategy` developed by TinkerGraph developers can optimize operations related to Gremlin API $count()$.

The complexity of Gremlin query's execution poses a major correctness challenge for GDBs, because the combination space of Gremlin traversals is huge. For a complex Gremlin query, incorrect implementations and optimizations of GDBs can introduce logic bugs. Our approach can prevent some optimization strategies from kicking in by disassembling a Gremlin query into a sequence of atomic graph traversals, thus detecting logic bugs in GDBs.

3 Approach

We propose *Query Disassembling* (`QuDi`), an effective approach for automatically finding logic bugs in Gremlin-based GDBs. `QuDi` solves the test oracle problem by disassembling a complex Gremlin query (i.e., an *original query*) into an equivalent atomic graph traversal sequence. We compare the query result of the original query with the query result of its corresponding disassembled atomic graph traversal sequence, and then identify the discrepancy between their query results as a logic bug.

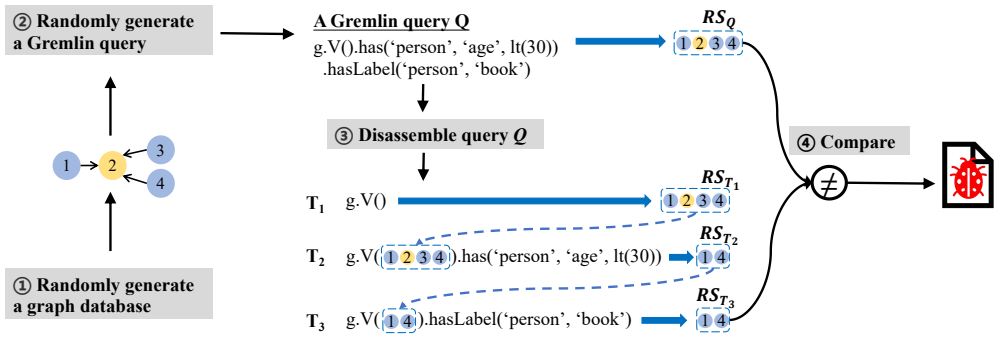

Figure 4: Approach overview.

Figure 4 illustrates the overview of our approach. We first randomly generate a graph database (①). The generated graph database consists of a set of vertices and a set of edges (e.g., the graph data shown in Figure 2). We then randomly generate Gremlin queries based on the traversal model proposed in Figure 3 and the graph database generated in the previous step (②). For each randomly generated Gremlin query Q (e.g., the query in Figure 1), we first execute it on the target GDB to compute a result set RS_Q (e.g., v:{1, 2, 3, 4}). After that, we disassemble Q into a sequence of atomic graph traversals $TList$ (e.g., $< T_1, T_2, T_3 >$), which has the equivalent execution semantics with the original query Q (③). For each atomic graph traversal T_i in $TList$, we construct an independent Gremlin query for it, which takes the result set $RS_{T_{i-1}}$ of its previous traversal T_{i-1} as input. We then execute T_i's query and compute its result set RS_{T_i}. After executing the last atomic graph traversal in $TList$ (e.g., T_3), we get a final result set (e.g., v:{1, 4}). Finally, we compare the result set RS_Q of the original query Q with the result set of $TList$ (e.g., RS_{T_3}) (④). A logic bug is reported for the target GDB if these two query results are different.

By disassembling a Gremlin query Q into an equivalent atomic graph traversal sequence $TList$, we can achieve the following two targets, and effectively test individual Gremlin-based GDBs.

- For a target GDB, the combination space of atomic graph traversals is huge. However, we cannot know whether an assembly of atomic graph traversals is correct or not. `QuDi` can utilize atomic graph traversals to construct a test oracle for complex Gremlin queries, and intensely test them without human intervention.
- During sequentially executing atomic graph traversals in $TList$, we disable some complex query optimization mechanisms in a target GDB for complex Gremlin queries and prevent optimizations from kicking in. Thus, we can construct a test oracle for query optimization in a target GDB.

3.1 Graph Database and Query Generation

In this work, we generate random graph databases and random Gremlin queries based on *Grand* [64], a differential testing framework for finding bugs in Gremlin-based GDBs. Here, we explain the graph database and query generation only for completeness.

Graph database generation. To generate a graph database, we first randomly generate a graph schema that defines the vertex and edge types. We denote a vertex type as $< label, propertyType* >$, in which $label$ and $propertyType*$ represent its label name and property types, respectively. An edge type can be represented as $< label, propertyType*, inVType, outVType >$, in which $label$ represents its label name, $propertyType*$ is a set of property types, and $inVType$ and $outVType$ denote its incoming and outgoing vertex types, respectively. A property type $propertyType$ contains a property name and a data type. Then, we can generate a vertex type by generating a random label name and a set of properties, each of which contains a random property name and a random data type. We can randomly generate a label name and a set of property types for an edge type, and randomly select a vertex type as its incoming (outgoing) vertex type.

Based on the generated graph schema, we can generate a set of vertex and edge instances, which are composed as a graph database. To generate a vertex, we first randomly choose a vertex type, and then randomly generate its property values. To generate an edge, we first randomly choose an edge type, and randomly choose two vertices whose vertex type satisfy its incoming and outgoing vertex types, respectively. The generation of its label name and properties is similar to the vertex generation.

We also create graph indexes for vertex (edge) properties, which can improve the query efficiency of those indexed vertices (edges). Specifically, we first randomly select some properties of vertices (edges) in the generated graph schema, and then we create graph indexes for these randomly chosen properties by using the specific syntaxes and index mechanisms of target GDBs.

Gremlin query generation. We randomly generate Gremlin queries based on the abstract traversal model in Figure 3. Given a graph traversal length L, we iteratively select traversal types, until L is reached or a `Value` type is selected. Specifically, in each iteration, we randomly select a traversal type (e.g., `Filter`), and then randomly choose a Gremlin API (e.g., $has()$) in the traversal model. Its output type can be transmitted to the next iteration, which guarantees the syntax correctness of a generated Gremlin query. Some Gremlin APIs require parameters, e.g., property names and property values. We can select a property name or a property value from the graph database to reduce the probability of an empty result set, or generate it with a Random function.

3.2 Query Disassembling

Given a generated Gremlin query, we disassemble it into a sequence of atomic graph traversals. An atomic graph traversal can achieve one-step traversal in the graph database and may contain one or

Algorithm 1: Query Disassembling

Input: Q (A Gremlin query)
Output: $TList$ (A sequence of atomic graph traversals)

```
1  TList ← {}
2  atomicT ← {}
3  APISequence ← getGremlinAPICalls(Q)
4  for i ← 1; i ≤ APISequence.length; i++ do
5  |   api ← APISequence[i]
6  |   atomicT.add(api)
7  |   if api.outType = VERTEX | EDGE then
8  |   |   TList.add(atomicT)
9  |   |   atomicT ← {}
10 end
11 if atomicT ≠ NULL then
12 |   TList.add(atomicT)
13 return TList
```

more Gremlin APIs. For example, $g.V()$, $has(lt())$, and $hasLabel()$ in Figure 4 can be treated as atomic graph traversals. Some Gremlin APIs cannot be treated as atomic graph traversals because they cannot be executed independently and cannot return any graph data. For example, in Figure 4, we cannot treat $lt()$ as an atomic graph traversal because $lt()$ is only used for filtering conditions and acts as a parameter in $has()$.

In our approach, we refer to an atomic graph traversal as a group of Gremlin API calls, which return a result set with an output type of `Vertex` or `Edge` (e.g., $g.V()$ and $g.E()$ in Figure 3). We further require that, if an atomic graph traversal returns a result set with an output type of `Vertex` or `Edge`, any of its sub-sequence of Gremlin API calls cannot return a result set with an output type of `Vertex` or `Edge`. Take $g.V().has('person','age',lt(30))$ as an example. We cannot treat this query as an atomic graph traversal, since both $g.V()$ and $has('person','age',lt(30))$ can return a result set of `Vertex`. Thus, we will disassemble this into two atomic graph traversals, i.e., $g.V()$ and $has('person','age',lt(30))$. Note that we also consider all the Gremlin API calls at the end of a Gremlin query to compute `Value` for `Vertex` or `Edge` as the last graph traversal. For example, we treat $values('age').sum()$ as the last graph traversal for query $g.V().values('age').sum()$.

Algorithm 1 illustrates how we disassemble a Gremlin query into a sequence of atomic graph traversals. Initially, we set an atomic graph traversal $atomicT$ to an empty list (Line 2). We then extract Gremlin API calls of the given Gremlin query Q from its traversal steps in turn (Line 3). For each Gremlin API call api, we add it to $atomicT$ (Line 6), and then check whether we can disassemble after it (Line 7-9). Specifically, we check whether the output type $outType$ of api is `Vertex` or `Edge`. If yes, we get an atomic graph traversal, and add $atomicT$ to the traversal list $TList$, and then continue to find a new atomic graph traversal. After processing all the Gremlin API calls in Q, we will add $atomicT$ to $TList$ if $atomicT$ is not empty (Line 11-12). Finally, we obtain the disassembled atomic graph traversal sequence $TList$ and return it (Line 13).

According to Algorithm 1, Gremlin APIs in `Vertex` in Figure 3 can return a vertex list, and thus can be treated as atomic graph traversals. For example, Gremlin API $outV()$ returns a list of outgoing vertices, and thus can be treated as an atomic graph traversal. Similarly, Gremlin APIs in `Edge` in Figure 3 can also be treated as atomic graph traversals. Gremlin APIs in `Filter` can also be treated as atomic graph traversals, because they can maintain the vertex or edge type of its previous graph traversal. For example, we can treat $hasLabel()$ as an atomic graph traversal.

```
1  g.V();  -- v:{1,2,3,4}
2  g.V(1,2,3,4).has('person','age',lt(30));  -- v:{1,4}
3  g.V(1,4).hasLabel('person','book');  -- v:{1,4}
```

Figure 5: The execution of the disassembled atomic graph traversals in Figure 4 using the parameter passing strategy.

Note that it is unnecessary to disassemble Gremlin APIs in `Value`. For example, for a query $values().count()$, we could not disassemble it into $values()$ and $count()$. Further, we do not disassemble Gremlin APIs in `Predicate` because they cannot return any vertices or edges, and always act as parameters for filtering operations.

3.3 Atomic Traversal Execution

For each atomic graph traversal T_i in the disassembled graph traversal sequence $TList$, we construct a Gremlin query Q_i for it to compute its intermediate result RS_{T_i}. Here, we refer to the constructed query as a disassembled query. To construct and execute the disassembled query Q_i for T_i, we need to firstly retrieve the intermediate result set $RS_{T_{i-1}}$ of its previous traversal T_{i-1} as input. The intermediate result $RS_{T_{i-1}}$ can be a list of vertices or edges. Note that for the start traversals $g.V()$ and $g.E()$, we do not need to construct additional queries for them. To correctly and effectively store and use these intermediate results, we come up with three execution strategies, i.e., *parameter passing strategy*, *temporary ID table strategy*, and *barrier strategy*. These three strategies execute the disassembled query using different Gremlin features, and can potentially find more logic bugs. Our experiment in Section 4.2.3 shows that these three strategies can complement each other.

3.3.1 Parameter Passing Strategy. In this strategy, we store the intermediate query result of an atomic graph traversal T_{i-1} into an ID list $idList$, which is a list of vertex IDs or edge IDs. When we compute the graph traversal T_i, we retrieve the intermediate result from $idList$, and pass it as a parameter of the Gremlin API $V(idList)$ (obtaining vertices with a vertex ID list) or $E(idList)$ (obtaining edges with an edge ID list) according to the output type of T_{i-1}. For example, as shown in Figure 5, we store the intermediate result of the second atomic graph traversal to a vertex list $\{1,4\}$ (Line 2). When we execute the third atomic graph traversal, we first retrieve the vertex list $\{1,4\}$ and then take it as the parameter of $V()$, i.e., $g.V(1,4)$. Finally, we construct a query by attaching the third atomic graph traversal $hasLabel('person','book')$ behind $g.V(1,4)$ to compute the final result (Line 3).

Note that when a graph traversal returns an empty list, we cannot pass an ID list to API $V()$ or $E()$. In this case, we construct a query that returns an empty list for $V()$ or $E()$. For example, we can generate a specific ID that does not exist in the graph database so that the constructed query cannot retrieve any vertex (or edge).

The parameter passing strategy can effectively disable complex query assembly and optimizations, so that we can detect logic bugs efficiently. But this strategy is limited by the size of the parameters in API $V()$ and $E()$, so that a large graph database is not suitable for this strategy. For example, since JanusGraph limits the size of ID list in $V()$ or $E()$ to 255, an exception will be thrown if more than 255

```
1  // put the result v:{1,4} into a temporary table
2  g.addV('IDs').property('id',1)
3  g.addV('IDs').property('id',4)
4
5  // execute the third atomic graph traversal
6  g.V().hasLabel('IDs').values('id').as('vList')
7   .V().as('V')
8   .id().as('V_ID')
9   .where('vList',P.eq('V_ID')).select('V')
10  .hasLabel('person','book')   -- v:{1,4}
11
12 // drop the temporary ID table
13 g.V().hasLabel('IDs').drop()
```

Figure 6: The execution of the atomic graph traversal T_3 in Figure 4 using the temporary ID table strategy.

vertices or edges are returned for an atomic graph traversal. Fortunately, we find that all the 25 detected logic bugs in our experiment can also be triggered with a small number of graph data, i.e., at most three vertices and two edges in a graph database. This indicates that the limitation about the size of parameters (255) should not affect the effectiveness of the parameter passing strategy.

3.3.2 Temporary ID Table Strategy. In this strategy, we store the intermediate result to a temporary table. Specifically, we first extract an ID list *idList* from the intermediate result $RS_{T_{i-1}}$ of T_{i-1}, and then store each ID in *idList* to a newly created vertex in the graph database. Thus, we can get a list of newly created vertices *vList* to store the ID list *idList* of $RS_{T_{i-1}}$. Specifically, each vertex in *vList* has a label 'IDs' with a property *id* to store an ID in *idList*. When we execute the graph traversal T_i, we construct a query to retrieve the result set of T_{i-1} with the help of *vList* and then execute T_i. After we consume these vertices, we will delete them.

As shown in Figure 6, we take the execution of the third atomic graph traversal *hasLabel('person',' book')* (T_3 in Figure 4) as an example. After we obtain the result set v:{1, 4} of the second atomic graph traversal T_2, we store their vertex IDs into a table with label 'IDs' (Line 2-3). Note that label 'IDs' can only be used in this part, and cannot be used in graph database generation and Gremlin query generation (Section 3.1). Next, we first query the vertex list *vList* (Line 6), and then retrieve vertices whose IDs are in the vertex list *vList* from all vertices in the graph database (Line 7-9). After that, we execute T_3 to compute its result set v:{1, 4} (Line 10). Finally, we remove the created vertices (Line 13).

Since the graph database generator does not generate vertices or edges whose label is 'IDs' to a randomly generated database, the newly created vertices do not affect the correctness of our approach. Although we construct a complex Gremlin query to retrieve the intermediate results, we can still break down the assembly of the original Gremlin query.

This strategy can avoid the disadvantages of parameter passing strategy, e.g., the limited size of the parameters in Gremlin API $V()$ and $E()$. Furthermore, the idea of storing the intermediate result to a temporary table can be commonly used in other database systems, e.g., it can be extended to disassemble SQL queries in relational database systems (Section 5). However, for this strategy, we must spend more time to test target GDBs, because it needs more operations to store or drop the intermediate results in the graph database. Besides, since we introduce extra atomic graph traversals to retrieve and use the intermediate results, bugs related to these traversals might be missed.

```
1  g.V().barrier().has('person','age',lt(30)).barrier().
       hasLabel('person','book');  -- v:{1,4}
```

Figure 7: The execution of the disassembled atomic graph traversals in Figure 4 using the barrier strategy.

3.3.3 Barrier Strategy. The core insight of this strategy is that the Gremlin API *barrier*() can turn the lazy pipeline of a Gremlin query into a bulk-synchronous pipeline. This indicates that the graph traversals prior to a *barrier*() operation need to be executed before moving onto the graph traversals after the *barrier*() operation. Therefore, when we append a *barrier*() operation after atomic graph traversal T_{i-1}, the assembly of T_{i-1} and T_i in the original Gremlin query can be disabled by the *barrier*() operation. Note that we do not need to append a *barrier*() operation after the last atomic graph traversal. For example, in Figure 7, we first append *barrier*() operations after the first and second atomic graph traversals to construct the disassembled query, and then execute it to compute the result set of the disassembled atomic graph traversal sequence.

Although this strategy can disable Gremlin API assembly and prevent some optimizations from kicking in, it introduces bulk operation and bulk optimization (when repeatedly touching many of the same elements, *barrier*() operations can only execute this element once) itself. As such, this strategy cannot disable more complex query assembly and optimizations as the other two execution strategies do, so that some logic bugs may be omitted. However, we find that one bug detected in our experiment by this strategy cannot be detected by the other two strategies. This indicates that the barrier strategy can complement the other two strategies.

4 Evaluation

We implement QuDi on Grand [64] with around 1000 lines of Java code. Furthermore, we add some general Gremlin APIs, e.g., *sample*() and *barrier*(), to the traversal model used in Grand and add some GDB-specific features, e.g., creating graph indexes for properties. We modify the Gremlin query generation algorithm used in Grand by appending Gremlin traversal steps instead of String values to a Gremlin traversal source g. Therefore, we can easily disassemble Gremlin queries based on the assembly Gremlin traversal steps.

To evaluate the effectiveness of our approach, we apply QuDi on six representative Gremlin-based GDBs and investigate the following three research questions:

- **RQ1:** How effective is QuDi in detecting logic bugs in real-world GDBs?
- **RQ2:** How do the proposed three execution strategies in QuDi perform in detecting logic bugs?
- **RQ3:** How does QuDi compare with the existing state-of-the-art approaches in bug detection capability?

We first introduce our experimental methodology (Section 4.1). Then, we elaborate an overview of found bugs (Section 4.2) and comparisons to existing approaches (Section 4.3). Finally, we discuss some interesting bugs we discovered in detail (Section 4.4).

4.1 Methodology

Target GDBs. We evaluate QuDi on six widely-used Gremlin-based GDBs, i.e., Neo4j [12], OrientDB [16], JanusGraph [8], HugeGraph [6], TinkerGraph [22], and ArcadeDB [14]. Table 1 shows their

Table 1: Target GDBs

GDB	Ranking	GitHub Stars	Initial Release
Neo4j	1	12.4k	2007
OrientDB	6	4.7k	2010
JanusGraph	12	5.1k	2017
HugeGraph	33	2.5k	2018
TinkerGraph	34	1.9k	2009
ArcadeDB	37	434	2021

basic information, including DB-Engines Ranking of GDBs [3], GitHub stars, and initial release date, which indicate that they are all important and popular GDBs.

Among the six GDBs, four (i.e., Neo4j, JanusGraph, HugeGraph, and TinkerGraph) only support the graph model, and the remaining two target GDBs, i.e., OrientDB and ArcadeDB, support multiple data models, e.g., document, graph, and key-value models. Furthermore, these GDBs support Gremlin APIs in different ways. We access Neo4j through the Neo4j-Gremlin plugin [13], which is provided by Apache TinkerPop[1]. JanusGraph, HugeGraph, and TinkerGraph encapsulate a Gremlin server in their own servers. They apply some special optimizations and can also natively use Gremlin to query graph data. OrientDB and ArcadeDB implement their own TinkerPop3 interfaces, so that they can use Gremlin to query graph data. Thus, the six GDBs used in our experiments can cover different kinds of Gremlin-based GDBs and are representative.

We test the latest release versions of these GDBs when we start this work, i.e., Neo4j 3.4.11 (with the latest release version Neo4j-Gremlin 3.6.1), OrientDB 3.2.10, JanusGraph 0.6.2, HugeGraph 0.12.0, TinkerGraph 3.6.1, and ArcadeDB 22.8.1.

Testing methodology. We run `QuDi` to test each target GDB with each execution strategy (i.e., parameter passing strategy, temporary ID table strategy, and barrier strategy) in 10 testing rounds. In each testing round, a graph database with at most 50 vertices and 100 edges is randomly created for a target GDB, and 1,000 Gremlin queries are randomly generated as original queries. Note that our approach can be applied to find bugs involving large graph data, e.g., 5,000 vertices and 10,000 edges. However, the parameter passing strategy cannot be used for large graph data (e.g., more than 255 vertices or edges). Thus, we only generate at most 50 vertices and 100 edges to make sure that all of our three execution strategies can work. All the numbers (i.e., 50 vertices, 100 edges, and 1,000 Gremlin queries) are configurable.

Simplify test cases. For each reported logic bug, we manually reduce the test case to a smaller one so that we can easily understand and diagnose the bug. The manual reduction is conducted as follows. (1) We remove the last unchecked traversal step in the original query, and check whether the same bug can be still triggered by the simpler test case. (2) If the bug can still be triggered, we have obtained a simpler test case. We will continue step (1) on the simpler test case for further reduction. (3) If the bug cannot be triggered, we will continue step (1) on the original test case. (4) We continue the above steps until we find a manageable and simpler test case.

Deduplicate test cases. For a simplified test case, we manually analyze its query pattern and bug consequences, and check whether it occurs in previous test cases. If yes, we consider it as a duplicate test case and discard it. Otherwise, we further check whether its possible root cause occurs in previous test cases. If yes, we also consider it as a duplicate test case. For example, we detect an issue that throws an exception *com.baidu.hugegraph.backend.id.Id* when executing the simplified query $g.V().outE(el1, el2).has('ep1')$. We analyze its root cause by trying to find which query pattern triggers this exception. In this example, we find that $has()$ cannot be queried after $outE()$ with multiple labels in $outE()$, the graph pattern $outE(el1, el2).has('ep1')$ triggers the exception. After that, if we find a simplified test case that contains $outE(a, b).has()$ and throws an exception *com.baidu.hugegraph.backend.id.Id*, then we consider it as a duplicate. Finally, we submit unique issues to the corresponding community on GitHub.

[1]We can test Neo4j and the Neo4j-Gremlin plugin at the same time.

4.2 Detected Bugs

4.2.1 Bug Overview. We obtain 3,047 bug reports in the six tested GDBs, which may be potential logic bugs. Specifically, there are 8, 20, 265, 2490, 87, and 177 bug reports in Neo4j, OrientDB, JanusGraph, HugeGraph, TinkerGraph, and ArcadeDB, respectively. We carefully reproduce and analyze test cases in these 3,047 bug reports, and deduplicate these test cases according to their query patterns or root causes (Section 4.1). Finally, we obtain 25 unique real-world logic bugs in the six target Gremlin-based GDBs. Table 2 shows the overall bug statistics.

At the time of writing this paper, out of the 25 detected bugs, 17 logic bugs have been confirmed by developers. Among these 17 confirmed bugs, 10 logic bugs are classified as previously-unknown bugs and 7 logic bugs are considered as duplicate to existing bug reports. For the 7 duplicate bugs, we generate different test cases that are different from ones in the existing bug reports. For now, 8 out of the 17 confirmed bugs have been fixed by GDB developers. 2 logic bugs are considered as intended by GDB developers. The remaining 6 logic bugs have not been confirmed yet.

Note that in *Neo4j*, we detect one logic bug. However, we carefully investigate it and find that this bug is caused by the Neo4j-Gremlin plugin instead of Neo4j itself. We further find that this bug is similar to the bug report we have submitted to the TinkerPop community earlier. Therefore, we do not generate a new bug report for it and do not count it in Table 2.

Intended bugs. One intended bug OrientDB#9885 is introduced because OrientDB forgets to throw an exception when sorting vertices or edges for a not existing property. Although OrientDB developers explained that OrientDB can return *NULL* when accessing a not existing property, we still think it is a true bug because OrientDB sometimes does not return *NULL*. In another intended bug HugeGraph#1966, HugeGraph returns an incorrect query result when querying vertices or edges by filtering properties using $not(eq())$. HugeGraph developers explained that HugeGraph does not support not-eq index queries now. Nevertheless, HugeGraph developers still plan to improve it in the future.

> `QuDi` *can effectively detect logic bugs, i.e., 10 out of 25 detected logic bugs have been confirmed as previously-unknown bugs.*

4.2.2 Bug Analysis. We analyze logic bugs detected by `QuDi` in the following two aspects, i.e., bug categories and bug consequences,

Table 2: Logic Bugs Detected by QuDi

GDB	Detected Bugs					Fixed	Execution Strategies		
	Detected	New	Duplicate	Intended	Unconfirmed		ParaPass	TempID	Barrier
Neo4j	0	0	0	0	0	0	0	0	0
OrientDB	3	1	1	1	0	1	2	3	3
JanusGraph	3	2	0	0	1	0	3	2	1
HugeGraph	16	6	4	1	5	4	15	12	11
TinkerGraph	1	0	1	0	0	1	1	1	0
ArcadeDB	2	1	1	0	0	2	2	2	2
Total	**25**	**10**	**7**	**2**	**6**	**8**	**23**	**20**	**17**

to give an intuition of whether these logic bugs can seriously affect the reliability of GDBs.

Bug categories. Among these 17 confirmed logic bugs, 6 logic bugs are caused by incorrect implementations of atomic graph traversals, in which the related atomic graph traversals return incorrect results. For example, in HugeGraph#1946, HugeGraph cannot support $E(idList)$ correctly when its parameter contains duplicate edge IDs. 11 bugs are caused by incorrect implementations of the assembly of atomic graph traversals (e.g., Figure 1). In these bugs, all their used atomic graph traversals can return correct results, but the assembly of the used atomic graph traversals returns an incorrect result. Among these 11 bugs, 3 bugs are related to the incorrect query optimizations of related assembly of atomic graph traversals. For example, in TinkerGraph#2812, TinkerGraph returns a wrong result for $order().by('prop').count()$ because its optimization mechanisms cannot properly process $count()$ operation.

Note that we can find logic bugs caused by the incorrect implementations and optimizations because QuDi can prevent some optimization strategies from kicking in by disassembling a complex Gremlin query into atomic graph traversals. Furthermore, we can also potentially detect logic bugs caused by incorrect implementations in atomic graph traversals. If an atomic graph traversal is incorrectly implemented, but a Gremlin query that contains this atomic graph traversal returns a correct result, QuDi can also identify the inconsistency and detect a logic bug.

Bug consequences. We summarize the 25 logic bugs into three categories according to their bug consequences, i.e., *incorrect query result*, *lacking exception*, and *unexpected exception*[2]. Table 3 shows the detailed information. Specifically, *incorrect query result* means the two compared results are not equal, and one of them is incorrect. 9 bugs belong to this category, including 4 previously-unknown bugs (one has been fixed), 3 duplicate bugs (two of them have been fixed) and two unconfirmed bugs. *Lacking exception* means that a Gremlin query should throw an exception (e.g., NumberFormatException) but returns a query result instead. 5 bugs are related to it, including 2 previously-unknown bugs (one has been fixed), one intended bug, and two unconfirmed bugs. *These 14 bugs that cause incorrect query result and lacking exception can easily go unnoticed by developers because no exception or warning is thrown.*

The remaining 11 bugs belong to *unexpected exception*, each of which returns an unexpected exception for a valid query, but they should not. In this case, users cannot get their expected results correctly. 8 out of 11 bugs have been confirmed, including 4 previously-unknown bugs (two of them have been fixed). Note that all these bugs return commonly-thrown exceptions, e.g., IllegalArgumentException, NumberFormatException, NoIndexException, and IllegalStateException, which can also be returned by invalid queries. *Thus, these bugs are also easily ignored by GDB developers and omitted by the existing GDB testing techniques.*

[2]We consider *lacking exception* and *unexpected exception* as logic bugs since they can return the incorrect results without crashing the GDBs.

Table 3: Bug Consequences

GDB	Incorrect Query Result	Lacking Exception	Unexpected Exception
Neo4j	0	0	0
OrientDB	0	1	2
JanusGraph	2	0	1
HugeGraph	4	4	8
TinkerGraph	1	0	0
ArcadeDB	2	0	0
Total	**9**	**5**	**11**

Logic bugs detected by QuDi *can lead to incorrect query results, lacking exceptions, and unexpected exceptions, which can easily go unnoticed by GDB developers.*

4.2.3 Execution Strategies Comparison. We design three execution strategies, i.e., parameter passing strategy (ParaPass for short), temporary ID table strategy (TempID for short), and barrier strategy (Barrier for short) for storing and using intermediate results of a sequence of atomic graph traversals, aiming at finding more logic bugs using different Gremlin features.

We count the logic bugs found by different execution strategies in the six target GDBs. Specifically, once we find a logic bug using an execution strategy, we will manually analyze whether the other two execution strategies can also detect it with this bug report. As shown in Table 2, almost all (23/25) bugs can be found by ParaPass, most (20/25) bugs can be detected by TempID, and 17 bugs can be found by Barrier. ParaPass and TempID can find more bugs than Barrier because they can disable atomic graph traversal assembly more thoroughly by storing vertex (edge) ID list into an array list or a temporary table. However, Barrier does not really store these intermediate results, and executes each atomic graph traversal by appending a $barrier()$ operation after this atomic graph traversal, which can affect the bug detection capability of Barrier.

We further analyze the necessary of these three execution strategies. As shown in Figure 8, according to the special Gremlin features, ParaPass and Barrier can find four and one new bugs that other two execution strategies cannot detect, respectively. Specifically, four bugs only detected by ParaPass are all triggered by passing the intermediate results as a parameter of the Gremlin API $V(idList)$ or $E(idList)$, which cannot be used in other two execution strategies. One bug only detected by Barrier is triggered by the feature of

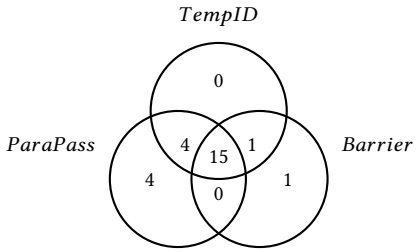

Figure 8: Venn diagram of the logic bugs detected by our three execution strategies.

API *barrier*(). `TempID` can find some logic bugs that are common to `ParaPass` and `Barrier`. However, `TempID` is a general strategy that can be potentially applied to test other database systems (Section 5), and can also work with large graph data. Overall, since these three execution strategies construct and execute the disassembled query using different Gremlin features, they have different bug detection capabilities, and thus do not detect exactly the same bugs. Thus, our three execution strategies can complement each other.

Our three execution strategies can cover different Gremlin features and complement each other, thus detecting more logic bugs.

4.3 Comparing with Existing Approaches

Four related approaches, i.e., Grand [64], GDsmith [39], RD^2 [62], and GDBMeter [42], can find bugs in GDBs. Specifically, Grand, GDsmith, and RD^2 utilize *differential testing* [49] to reveal discrepancies among multiple target GDBs. GDBMeter utilizes Ternary Logic Partitioning (TLP), an invariant of *query partitioning* [52], to reveal logic bugs in a target GDB. Therefore, we compare `QuDi` with *differential testing* and *query partitioning*. Note that we cannot compare `QuDi` with relational database testing tools, e.g., SQLancer [51–53], since `QuDi` and SQLancer target different types of database systems, which utilize different query languages (Gremlin vs. SQL) and data models (graph model vs. relational model).

Comparison with differential testing. Differential testing (e.g., Grand) requires more than one target GDB as input. If a Gremlin feature is not supported by all target GDBs, differential testing will not be able to apply on this feature. Thus, differential testing can only be used to test common features in the target GDBs. We can encounter the following cases in differential testing.

- **C1**: At least two GDBs can return inconsistent results, then differential testing reports a bug.
- **C2**: All GDBs return the same wrong results, then differential testing can miss a bug.
- **C3**: All GDBs return the same results. However, some GDBs are expected to return inconsistent results according to their special semantics. In this case, differential testing can miss a bug.
- **C4**: At least two GDBs can return inconsistent results. However, the inconsistency is caused by target GDBs' special semantics. In this case, differential testing can report a false positive.

Based on the above discussion, `QuDi` has the following advantages over differential testing. (1) `QuDi` can detect bugs based on an individual GDB. (2) `QuDi` can test Gremlin features that are supported by only one GDB. (3) `QuDi` is used to detect internal inconsistencies in an individual GDB, and will not encounter C3 and C4.

To compare `QuDi` with differential testing, we first verify whether differential testing in Grand can detect the 25 logic bugs detected by `QuDi`. We run each test case in our 25 bug reports on our six target GDBs and verify whether their query results are the same. Any discrepancy among their query results will be considered as a logic bug detected by Grand. We find that, Grand can detect 19 logic bugs that `QuDi` find, but cannot detect the remaining 6 bugs. Besides, Grand reports 2 false positives because the buggy GDBs return different query results from the other GDBs but their behaviors are expected due to the GDBs' special semantics.

Note that it is reasonable that Grand can detect many logic bugs that `QuDi` can detect, since Grand use multiple GDBs as reference implementations. It is interesting to know how effective `QuDi` is in revealing logic bugs detected by Grand. We further verify whether `QuDi` can detect the 21 logic bugs reported by Grand. Specifically, for an original Gremlin query that triggers a bug in these 21 bug reports, we disassemble it into atomic graph traversals and verify whether they can compute different query results as the original query. If yes, we consider that `QuDi` can detect the logic bug. We find that, `QuDi` can find 16 (76%) logic bugs detected by Grand. `QuDi` misses the remaining 5 bugs because their original queries and the corresponding disassembled queries return the same wrong results. *This result indicates that* `QuDi` *has the capability to detect most logic bugs detected by Grand without suffering from Grand's drawbacks, e.g., multiple GDBs as input and false positives.*

Comparison with query partitioning. In query partitioning (e.g., GDBMeter), a given query Q can derive multiple disjoint sub-queries (e.g., $Q'_{p=TRUE}$, $Q'_{p=FALSE}$, and $Q'_{p\ is\ NULL}$ based on a random predicate p in TLP). These individual partitions are composed to obtain a result set $RS_{Q'}$, which should be equal to RS_Q. However, query partitioning cannot prevent GDB optimizations from kicking in and reveal assembly issues in GDBs, so that it can hardly detect logic bugs that are caused by incorrect implementations and optimizations of the assembly of atomic graph traversals, and our query disassembling is complementary to query partitioning.

We first verify whether query partitioning in GDBMeter can detect the 25 logic bugs detected by `QuDi`. For each test case in our 25 bug reports, we try to construct disjoint sub-queries according to the TLP oracle and check whether these disjoint sub-queries can report an inconsistency. We find that, only 6 bugs detected by `QuDi` can be found by GDBMeter theoretically. Among these six bugs, 5 bugs are identified by unexpected exceptions in disjoint sub-queries and the remaining one is identified by incorrect query results. The remaining 19 bugs could not be detected by GDBMeter because the original queries in these test cases can compute the same wrong query results as the union query results of their disjoint sub-queries.

We further verify whether `QuDi` can detect the 3 logic bugs reported by GDBMeter in JanusGraph, which adopt Gremlin as the query language. We ignore the other 36 bugs reported by GDBMeter since their test cases use Cypher queries, which we do not support for now. We try to run each test case in these 3 bug reports via `QuDi`, and check whether `QuDi` can reveal them. However, 2 of 3 bugs are internal errors that are triggered by query generation instead of the test oracles (e.g., query partitioning and query disassembling).

```
1  v1 = g.addV('vL').next();
2  v2 = g.addV('vL').next();
3  e1 = g.addE('eL').from(v1).to(v2).next();
4  g.E(e1).property('p', new Float(0.94461));
5
6  // original query
7  g.E().has('p', 0.94461);  -- e:{1} ✘
8
9  // disassembled queries
10 g.E();  -- e:{1}
11 g.E(1).has('p', 0.94461);  -- {} ✔
```

Figure 9: JanusGraph inconsistently retrieves properties without explicit data type in JanusGraph#3200.

```
1  v1 = g.addV('vL').property('p0', 1).next();
2  v2 = g.addV('vL').property('p1', 2).next();
3
4  // original query
5  g.V().order().by('p0');  -- {The property does not exist as
       the key has no associated value for the provided
       element v[2]:p0.} ✘
6
7  // disassembled queries
8  g.V(); -- v:{1, 2}
9  g.V(1,2).order().by('p0');  -- v:{1,2} ✘
```

Figure 10: JanusGraph incorrectly sorts vertices with properties in JanusGraph#3216 and JanusGraph#3269.

Since we cannot reproduce the remaining one bug, we do not know whether QuDi can reveal it.

> QuDi *can find new bugs that existing approaches cannot detect.* QuDi *also has the capability to detect most (76%) logic bugs detected by Grand without suffering from Grand's drawbacks.*

4.4 Selected Interesting Bugs

We explain some interesting bugs found by QuDi to give an intuition of what kinds of bugs can be found via QuDi.

Inconsistent handling of data types. JanusGraph suffers from inconsistent behaviors when querying properties without giving explicit data types. As shown in Figure 9, we first create two vertices and one edge (Line 1-3). We then add a property p with a value 0.94461 whose type is *Float* to edge e:1 (Line 4). When we query edges by filtering $p = 0.94461$, we can retrieve edge e:1 with the original query of Line 7. However, when we disassemble it into two atomic graph traversals $g.E()$ and $has('p', 0.94461)$ (Line 10-11), an empty set is returned, which is inconsistent with the result of the original query. JanusGraph developers have confirmed this bug and are trying to investigate the root cause of this issue.

Incorrect handling of order() operation. We find two bugs where JanusGraph mistakenly sorts vertices or edges with properties using $order().by()$. As shown in Figure 10, two vertices v:1 and v:2 are created with property $p0$ and $p1$, respectively (Line 1-2). When we sort vertices with property $p0$ (Line 5), an exception is thrown because vertex v:2 does not have a property $p0$. But we can get a result set v:{1, 2} for the disassembled query (Line 9). For the bug triggered by the original query, JanusGraph developers think they should filter vertices based on property $p0$ instead of throwing an exception. The disassembled query triggers another bug, in which the expected result is v:{1} instead of v:{1, 2}. Although this bug has not been confirmed, we still believe it is a true bug.

```
1  v1 = new Vertex('vL');
2  v2 = new Vertex('vL');
3  v3 = new Vertex('vL');
4  e1 = v1.addEdge('eL', v2);
5  e2 = v1.addEdge('eL', v3);
6
7  // original query
8  g.V().bothE().count();  -- 4 ✔
9
10 // disassembled queries
11 g.V();  -- v:{1,2,3}
12 g.V(1,2,3).bothE();  -- e:{1,2,1,2}
13 g.E(1,2,1,2).count();  -- 2 ✘
```

Figure 11: HugeGraph retrieves wrongly edges when $E()$'s parameter contains duplicate edge IDs in HugeGraph#1946.

```
1  // original query
2  MATCH (p:Person)-[:Write]-> (b1:Book), (p)-[:Read]->(b2:Book
       ) RETURN b2  // -- {}
3
4  // disassembled queries
5  MATCH (p:Person)-[:Write]-> (b1:Book) RETURN p.id //--v:{1}
6  MATCH (p:Person)-[:Read]->(b2:Book) WHERE p.id=1 RETURN b2
       //--{}
```

Figure 12: An example of extending QuDi to Cypher query language.

Incorrect handling of E() operation. Figure 11 shows an example of the incorrect implementation of Gremlin API $E()$ in HugeGraph. In this test case, we create three vertices (i.e., v:1, v:2, and v:3) and two edges (i.e., e:1 and e:2) (Line 1-5). We want to count the number of the incoming and outgoing edges of all vertices (Line 8). The expected result is 4 because each edge is retrieved twice. However, when we execute the disassembled query of Line 13, a wrong result 2 is returned. Specially, the query $g.E(1, 2, 1, 2)$ returns an edge list e:{1, 2} instead of e:{1, 2, 1, 2}, which causes the incorrect result. HugeGraph developers have confirmed this bug.

5 Discussion

Generalizing QuDi to other database systems. By disassembling a complex query into multiple atomic queries, we can prevent some optimizations from kicking in, thus revealing logic bugs caused by the assembly of atomic queries and related optimizations. In this paper, we apply the idea of *query disassembling* only on Gremlin-based GDBs. However, query disassembling is a general idea, and can be potentially applicable on other database systems.

First, some other graph query languages, e.g., Cypher [24] and SPARQL [18], adopt the same philosophy as Gremlin (constructing a subgraph through a graph traversal model) to query GDBs. Therefore, we can also disassemble graph queries written in these graph query languages into multiple atomic queries through query disassembling. Figure 12 shows such an example, in which we disassemble a Cypher query (Line 2) into a sequence of atomic queries (Line 5-6) using query disassembling. Therefore, we can extend QuDi on those GDBs that support other graph query languages.

Second, query disassembling can also be applied to disassemble sub-queries and joins of SQL queries in relational database systems. As shown in Figure 13, we can disassemble the SQL query (Line 2) into two atomic queries. Specifically, one atomic query executes the sub-query (SELECT c1, c2 FROM t1) and stores its query results into a newly created table $t2$ (Line 5) and the other atomic query

```
1 // original query
2 SELECT c1 FROM (SELECT c1, c2 FROM t1);
3
4 // disassembled queries
5 CREATE TABLE t2 AS SELECT c1, c2 FROM t1;
6 SELECT c1 FROM t2;
```

Figure 13: An example of extending QuDi to SQL queries in relational database systems.

searches $c1$ from the newly created table $t2$ (Line 6). Thus, QuDi can also help to test relational database systems.

Limitations. Three kinds of bugs cannot be found by QuDi. First, QuDi cannot detect bugs in the Gremlin query APIs (e.g., $path()$ and $tree()$), whose outputs are affected by the entire execution of a Gremlin query. Second, QuDi cannot detect bugs in the Gremlin APIs related to vertex computations in Gremlin, such as $pageRank()$ and $shortestPath()$. These APIs perform more complex operations, which cannot be disassembled directly. Third, If the original query and its disassembled queries retrieve the same wrong query result, QuDi cannot detect it.

Threats to validity. First, we evaluate QuDi on six Gremlin-based GDBs. These GDBs rank on the top of GDB popularity, and all of them are well maintained. Thus, we believe they are representative for Gremlin-based GDBs. Second, we manually reproduce and deduplicate bug reports, which may introduce human errors. To alleviate this threat, three authors study all reported bugs carefully and reach a consensus for them. Third, the comparisons between QuDi and the closest-related works are also threats. Although we do not compare QuDi with GDBMeter by actual evaluation, we have made a careful analysis for them.

6 Related Work

Differential testing. Differential testing [49] is a common approach for finding bugs in several research domains [29, 49, 55, 61, 64]. Some approaches [30, 41, 46, 55] have been proposed to test relational database systems via differential testing. For example, APOLLO [41] detects performance regression bugs with differential testing. DT^2 [30] utilizes differential testing to find transaction bugs in relational database systems. To test GDBs, Grand [64], GDsmith [39], and RD^2 [60] utilize differential testing to find logic bugs in multiple GDBs with Gremlin, Cypher, and SPARQL query languages, respectively. However, differential testing cannot provide a testing oracle for a target GDB, and can miss logic bugs and encounter false positives due to target GDBs' special semantics.

Metamorphic testing. Metamorphic testing [48] aims to address the test oracle problem [28] by mutating test cases according to metamorphic relations. Some metamorphic testing approaches [37, 51, 52, 56, 57] for finding bugs in relational database systems have been proposed. For example, query partitioning [52] derives a given query to multiple disjoint sub-queries, in which the result of the given query is the same as the combination result of disjoint sub-queries. NoREC [51] transforms an optimized SQL query to a non-optimized SQL query, and can find optimization bugs in relational database systems. However, all the above approaches target database systems with *declarative* SQL query language. They cannot effectively test Gremlin-based GDBs because Gremlin and SQL have totally different syntaxes and query patterns.

To test GDBs, GDBMeter [42] derives a graph query into three disjoint sub-queries by randomly generating a predicate, which mainly focuses on predicate-related bugs. Gamera [65] designs three types of graph-aware metamorphic relations, which can be used to generate diverse and complex graph queries, and then reveal logic bugs in GDBs. Mang et al. [47] propose equivalent query rewriting (EQR), which rewrites a graph query into equivalent graph queries that trigger distinct query plans, to detect bugs in GDBs. Given a graph query Q, GraphGenie [40] derives a mutated graph query whose query result set is either semantically equivalent to the result set of Q or constitutes a subset or superset of the result set of Q, depending on the mutation applied. DOT [63] can detect optimization bugs by testing a graph query with two different optimization configurations. However, these approaches cannot reveal assembly issues in GDBs. Our approach proposes a new metamorphic rule to detect logic bugs related to incorrect implementations and optimizations of the assembly of atomic graph traversals. Thus, our approach is complementary to these approaches.

Other testing approaches of relational database systems. SQLsmith [19] can test relational database systems by randomly generating SQL queries. The generic fuzzing tools (e.g., AFL [1]) can also be used to test relational database systems. ADUSA [43, 44] uses a relational constraint solver, Alloy [2], to generate the expected result of a given SQL query. PQS [53] tests the correctness of relational database systems by randomly selecting a pivot row and generating random SQL queries that contain the selected row. QPG [27] can improve testing efficiency by exploring a variety of unique query plans. Troc [34] detects transaction bugs [31] by inferring the expected results of concurrent transaction executions. However, these approaches cannot be directly applied to test GDBs.

7 Conclusion

Buggy implementations and optimizations of Gremlin-based graph database systems can introduce logic bugs, which can lead to severe consequences, e.g., incorrect query results. In this paper, we propose query disassembling (QuDi) to reveal logic bugs in graph database systems by disassembling a complex Gremlin query into an equivalent atomic graph traversal sequence. We evaluate QuDi on six widely-used graph database systems, and have detected 25 unique logic bugs, 10 of which are previously-unknown bugs. We expect that the effectiveness and generality of our technique can greatly improve the robustness of graph database systems and other database systems.

8 Data Availability

The source code of QuDi is available at Zenodo [20].

Acknowledgments

We would like to thank the anonymous reviewers for their thorough and insightful comments. This work was partially supported by National Natural Science Foundation of China (62072444, 62302493), Major Project of ISCAS (ISCAS-ZD-202302), Major Program (JD) of Hubei Province (2023BAA018), Youth Innovation Promotion Association at Chinese Academy of Sciences (Y2022044, 2023121), and Guangdong Power grid limited liability company under Project 037800KC23090006.

References

[1] 2024. AFL. https://github.com/google/AFL.
[2] 2024. Alloy. https://alloytools.org/.
[3] 2024. DB-Engines Ranking of Graph DBMS. https://db-engines.com/en/ranking/graph+dbms.
[4] 2024. Gremlin Query Language. https://tinkerpop.apache.org/gremlin.html.
[5] 2024. Gremlin Traversal Strategy. https://tinkerpop.apache.org/docs/3.5.2/.
[6] 2024. HugeGraph. https://hugegraph.github.io/hugegraph-doc/.
[7] 2024. Introducing the new Cypher Query Optimizer. https://neo4j.com/blog/introducing-new-cypher-query-optimizer/.
[8] 2024. JanusGraph. https://janusgraph.org.
[9] 2024. MariaDB. https://mariadb.org.
[10] 2024. MySQL. https://www.mysql.com.
[11] 2024. Nebula Graph Query Language (nGQL). https://docs.nebula-graph.io/2.0.1/3.ngql-guide/1.nGQL-overview/1.overview/.
[12] 2024. Neo4j. https://neo4j.com/.
[13] 2024. Neo4j-Gremlin. https://github.com/thinkaurelius/neo4j-gremlin-plugin.
[14] 2024. The Next Generation Multi-Model Database Supporting Graphs Key/Value, Documents and Time-Series. https://arcadedb.com/.
[15] 2024. Open Source, Distributed, Scalable, Lightning Fast. https://nebula-graph.io/.
[16] 2024. OrientDB. https://orientdb.org.
[17] 2024. PostgreSQL. https://www.postgresql.org.
[18] 2024. SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/.
[19] 2024. SQLsmith. https://github.com/anse1/sqlsmith.
[20] 2024. *Testing Gremlin-Based Graph Database Systems via Query Disassembling*. Retrieved July 18, 2024 from https://doi.org/10.5281/zenodo.12771889
[21] 2024. TiDB, PingCAP. https://pingcap.com.
[22] 2024. TinkerGraph. https://github.com/tinkerpop/blueprints/wiki/tinkergraph.
[23] 2024. TinkerPop. https://tinkerpop.apache.org/.
[24] 2024. What is openCypher? http://www.opencypher.org/.
[25] Renzo Angles, Juan L. Reutter, and Hannes Voigt. 2019. Graph Query Languages. In *Encyclopedia of Big Data Technologies*, Sherif Sakr and Albert Y. Zomaya (Eds.).
[26] Marcelo Arenas, Claudio Gutiérrez, and Juan F. Sequeda. 2021. Querying in the Age of Graph Databases and Knowledge Graphs. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 2821–2828. https://doi.org/10.1145/3448016.3457545
[27] Jinsheng Ba and Manuel Rigger. 2023. Testing Database Engines via Query Plan Guidance. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 2060–2071. https://doi.org/10.1109/ICSE48619.2023.00174
[28] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525.
[29] Shafiul Azam Chowdhury, Soumik Mohian, Sidharth Mehra, Siddhant Gawsane, Taylor T. Johnson, and Christoph Csallner. 2018. Automatically Finding Bugs in a Commercial Cyber-Physical System Development Tool Chain with SLforge. In *Proceedings of International Conference on Software Engineering (ICSE)*. 981–992. https://doi.org/10.1145/3180155.3180231
[30] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. 2022. Differentially Testing Database Transactions for Fun and Profit. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. 35:1–35:12. https://doi.org/10.1145/3551349.3556924
[31] Ziyu Cui, Wensheng Dou, Yu Gao, Dong Wang, Jiansen Song, Yingying Zheng, Tao Wang, Rui Yang, Kang Xu, Yixin Hu, Jun Wei, and Tao Huang. 2024. Understanding Transaction Bugs in Database Systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. 163:1–163:13. https://doi.org/10.1145/3597503.3639207
[32] Alin Deutsch. 2018. Querying Graph Databases with the GSQL Query Language. In *Proceedings of Simpósio Brasileiro de Banco de Dados (SBBD)*. 313.
[33] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2020. Aggregation Support for Modern Graph Analytics in TigerGraph. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 377–392. https://doi.org/10.1145/3318464.3386144
[34] Wensheng Dou, Ziyu Cui, Qianwang Dai, Jiansen Song, Dong Wang, Yu Gao, Wei Wang, Jun Wei, Lei Chen, Hanmo Wang, Hua Zhong, and Tao Huang. 2023. Detecting Isolation Bugs via Transaction Oracle Construction. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 1123–1135. https://doi.org/10.1109/ICSE48619.2023.00101
[35] Orri Erling, Alex Averbuch, Josep Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 619–630. https://doi.org/10.1145/2723372.2742786
[36] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1433–1445. https://doi.org/10.1145/3183713.3190657
[37] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*. 345–358.
[38] Zhenzhen He, Jiong Yu, and Binglei Guo. 2022. Execution Time Prediction for Cypher Queries in the Neo4j Database Using a Learning Approach. *Symmetry* 14, 1 (2022), 55.
[39] Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, and Tao Xie. 2023. GDsmith: Detecting Bugs in Cypher Graph Database Engines. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*. 163–174. https://doi.org/10.1145/3597926.3598046
[40] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland H. C. Yap, Zhenkai Liang, and Manuel Rigger. 2024. Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. 46:1–46:12. https://doi.org/10.1109/ICST60714.2024.00012
[41] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woon-Hak Kang. 2019. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *Proceedings of the VLDB Endowment (VLDB)* 13, 1 (2019), 57–70.
[42] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. 2023. Testing Graph Database Engines via Query Partitioning. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3597926.3598044
[43] Shadi Abdul Khalek, Bassem Elkarablieh, Yai O. Laleye, and Sarfraz Khurshid. 2008. Query-Aware Test Generation Using a Relational Constraint Solver. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 238–247. https://doi.org/10.1109/ASE.2008.34
[44] Shadi Abdul Khalek and Sarfraz Khurshid. 2010. Automated SQL query generation for systematic testing of database engines. In *International Conference on Automated Software Engineering (ASE)*. 329–332. https://doi.org/10.1145/1858996.1859063
[45] Baozhu Liu, Xin Wang, Pengkai Liu, Sizhuo Li, Qiang Fu, and Yunpeng Chai. 2021. UniKG: A Unified Interoperable Knowledge Graph Database System. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*. 2681–2684. https://doi.org/10.1109/ICDE51399.2021.00303
[46] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems Using Equivalent Queries. In *Proceedings of International Conference on Software Engineering (ICSE)*. 225–236. https://doi.org/10.1145/3510003.3510093
[47] Qiuyang Mang, Aoyang Fang, Boxi Yu, Hanfei Chen, and Pinjia He. 2024. Testing Graph Database Systems via Equivalent Query Rewriting. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. 143:1–143:12. https://doi.org/10.1145/3597503.3639200
[48] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic Testing of Datalog Engines. In *Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 639–650. https://doi.org/10.1145/3468264.3468573
[49] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107.
[50] Yuxiang Ren, Hao Zhu, Jiawei Zhang, Peng Dai, and Liefeng Bo. 2021. EnsemFDet: An Ensemble Approach to Fraud Detection based on Bipartite Graph. In *Proceedings of International Conference on Data Engineering (ICDE)*. 2039–2044. https://doi.org/10.1109/ICDE51399.2021.00197
[51] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1140–1152. https://doi.org/10.1145/3368089.3409710
[52] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. 4, Article 211 (2020), 30 pages.
[53] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 667–682.
[54] Marko A. Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language (Invited Talk). In *Proceedings of the Symposium on Database Programming Languages (DBPL)*. 1–10. https://doi.org/10.1145/2815072.2815073
[55] Donald S. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*. 618–622.
[56] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing Database Systems via Differential Query Execution. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*. 2072–2084. https://doi.org/10.1109/ICSE48619.2023.00175
[57] Jiansen Song, Wensheng Dou, Yu Gao, Ziyu Cui, Yingying Zheng, Dong Wang, Wei Wang, Jun Wei, and Tao Huang. 2024. Detecting Metadata-Related Logic Bugs in Database Systems via Raw Database Construction. *Proceedings of the VLDB Endowment (VLDB)* (2024).

[58] Ran Wang, Zhengyi Yang, Wenjie Zhang, and Xuemin Lin. 2020. An Empirical Study on Recent Graph Database Systems. In *Proceedings of International Conference on Knowledge Science, Engineering and Management (KSEM)*. 328–340. https://doi.org/10.1007/978-3-030-55130-8_29

[59] Min Wu, Xinglu Yi, Hui Yu, Yu Liu, and Yujue Wang. 2022. Nebula Graph: An Open Source Distributed Graph Database. *CoRR* abs/2206.07278 (2022).

[60] Rui Yang, Yingying Zheng, Lei Tang, Wensheng Dou, Wei Wang, and Jun Wei. 2023. Randomized Differential Testing of RDF Stores. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE Demo)*. 136–140. https://doi.org/10.1109/ICSE-Companion58688.2023.00041

[61] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of International Conference on Programming Language Design and Implementation (PLDI)*. 283–294. https://doi.org/10.1145/1993316.1993532

[62] Yui Yang, Yingying Zheng, Lei Tang, Wensheng Dou, Wei Wang, and Jun Wei. 2023. Randomized Differential Testing of RDF Stores. In *Proceedings of International Conference on Software Engineering: Companion Proceedings (ICSE Companion)*. 136–140. https://doi.org/10.1109/ICSE-Companion58688.2023.00041

[63] Yingying Zheng, Wensheng Dou, Lei Tang, Ziyu Cui, Jiansen Song, Ziyue Cheng, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2024. Differential Optimization Testing of Gremlin-Based Graph Database Systems. In *Proceedings of the 17th IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 25–36.

[64] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. 2022. Finding Bugs in Gremlin-Based Graph Database Systems via Randomized Differential Testing. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*. 302–313. https://doi.org/10.1145/3533767.3534409

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