

# Effective Bug Detection in Graph Database Engines: An LLM-based Approach

Jiayi Wu  
Beijing Institute of Technology  
Beijing, China  
wjybit97@outlook.com

Zhengyu Wu  
Beijing Institute of Technology  
Beijing, China  
jeremywzy96@outlook.com

Rong-Hua Li  
Beijing Institute of Technology  
Beijing, China  
lironghuabit@126.com

Hongchao Qin  
Beijing Institute of Technology  
Beijing, China  
qhc.neu@gmail.com

Guoren Wang  
Beijing Institute of Technology  
Beijing, China  
wanggrbit@gmail.com

## ABSTRACT

Graph database engines play a pivotal role in efficiently storing and managing graph data across various domains, including bioinformatics, knowledge graphs, and recommender systems. Ensuring data accuracy within graph database engines is paramount, as inaccuracies can yield unreliable analytical outcomes. Current bug-detection approaches are confined to specific graph query languages, limiting their applicabilities when handling graph database engines that use various graph query languages across various domains. Moreover, they require extensive prior knowledge to generate queries for detecting bugs. To address these challenges, we introduce DGDB, a novel paradigm harnessing large language models (LLM), such as ChatGPT, for comprehensive bug detection in graph database engines. DGDB leverages ChatGPT to generate high-quality queries for different graph query languages. It subsequently employs differential testing to identify bugs in graph database engines. We applied this paradigm to graph database engines using the Gremlin query language and those using the Cypher query language, generating approximately 4,000 queries each. In the latest versions of Neo4j, Agensgraph, and JanusGraph databases, we detected 2, 5, and 3 wrong-result bugs, respectively.

## PVLDB Reference Format:

Jiayi Wu, Zhengyu Wu, Rong-Hua Li, Hongchao Qin, and Guoren Wang. Effective Bug Detection in Graph Database Engines: An LLM-based Approach. PVLDB, 14(1): XXX-XXX, 2020.  
doi:XX.XX/XXX.XX

## PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://anonymous.4open.science/r/A-ChatGPT-based-paradigm-for-detecting-bugs-in-graph-database-engines-3C56/>.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing [info@vldb.org](mailto:info@vldb.org). Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.  
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.  
doi:XX.XX/XXX.XX

## 1 INTRODUCTION

In the contemporary information age, a vast volume of data is generated and accrued, frequently characterized by intricate relationships and connections, and such data are often represented as graph structures. While relational databases are designed based on a tabular model[9] with Schema Rigidity, making it difficult to handle complex graph data. In response to this constraint, graph database engines have been designed to facilitate the storage and retrieval of graph data. They use the structure of nodes and edges to efficiently represent and process semi-structured data such as various relationships and networks[4, 33, 39]. Graph database engines like Neo4j[22], JanusGraph[2], AgensGraph[21], TinkerGraph[15], OrientDB[12], and HugeGraph[14], which rank prominently in the DB-Engine Ranking list[44], are widely utilized across various domains. For instance, on social media platforms such as Facebook and Twitter, graph database engines are employed to manage and analyze social relationships and behavioral patterns among users, aiding in the detection of fake users and online fraudulent activities[34]. In recommender systems, the utilization of graph database engines can dynamically generate real-time recommended content based on users' most recent actions and social connections[26, 42]. In the field of logistics and freight transportation, graph database engines can be harnessed to determine optimal delivery routes, thereby reducing costs and enhancing efficiency[11, 18].

As shown in Table 1, existing methods for detecting bugs in graph database engines can be broadly divided into two categories: methods for utilizing various queries to detect bugs within the same graph database engines, methods applying the same query on various graph database engines. The former typically involves detecting bugs in graph database engines by generating equivalent queries[5] or utilizing predicate partitioning[23, 24, 37]. The latter approach entails running the same queries on different graph database engines and identifying bugs based on discrepancies in query results[20, 50]. The former method demands users with substantial prior knowledge on the graph model and query objectives when generating equivalent queries to identify bugs in the graph database engine[5]. In the predicate partitioning approach for detecting graph database engine bugs[24], it starts by constructing queries based on a top-down expression generator[35], and then apply Ternary Logic Partitioning techniques to create correspondent queries, namely the "True, False or Null" queries, for running on graph database engine. Such a method assess whether the subset

**Table 1: Comparison of the proposed DGDB with closely related works.**

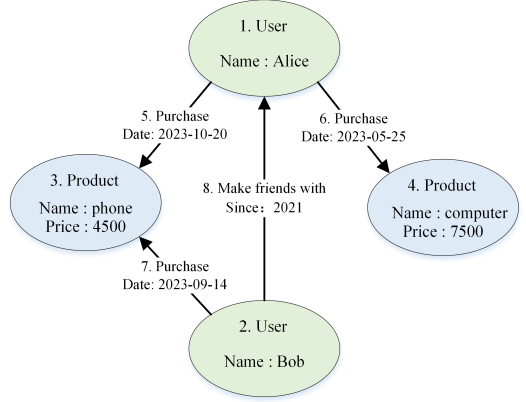
Method	Category	Language	Not require large prior knowledge	High proportion of non-empty-result queries
Grand[50]	the method applying the same query on various graph database engines	gremlin	✗	✗
GDSmith[20]	the method applying the same query on various graph database engines	cypher	✗	✓
PP-DB[24]	the method for utilizing various queries to detect bugs within the same graph database engines	all	✗	✗
DGDB	the method applying the same query on various graph database engines	all	✓	✓

of the final query results intersect to detect graph database engine bugs. While this approach successfully identifies some logical bugs, its effectiveness is limited due to the lower quality of queries generation and the manual need for reducing test samples.

Most current research focuses on the latter approach and the most widely adopted method is running randomly generated queries [3, 35, 43] on different graph database engines for bug detection. However, the generated queries often return empty results in many samples, which compromise the efficiency for bug detection. To address this issue, Grand[50] proposed a method based on traversal models to generate gremlin queries[40], significantly increasing the non-empty-result query ratio. However, this method is only applicable to gremlin-based graph database engines and requires users to master the gremlin API[41]. GDsmith[20] proposed a method to generate cypher queries[16] based on the guidance of property graphs. This method first generates a cypher skeleton, then creates patterns based on graph information, and ensures, through static query analysis, that the generated queries have high non-empty-result query ratios, thereby improving the efficiency of bug detection. However, this method is only applicable to cypher-based graph database engines and requires proficient mastery of cypher language.

Large language models(LLM)[49], as one of the most popular research directions in the field of artificial intelligence, are trained on massive text data, enabling models to understand and generate diverse text content, fostering innovation in many domains. For instance, OpenAI’s ChatGPT model[31], which is currently the most widely used large language model, is used in the recommendation domain to enhance text content, combining text with original content to improve the effectiveness of recommendation[28]; in the financial domain, it is used to infer network structures from text data and then integrate them with graph neural networks[47] to effectively predict stock trends[8]. Meta proposed the SAM model[25, 29], which can output specified image regions based on prompt information and has been applied in the fields of medical image segmentation and video object segmentation. Peking University’s AIGC Lab introduced the ChatLaw model[10], which not only provides legal consulting services to the general public but also serves as an assistant to professional lawyers.

Based on ChatGPT’s natural language understanding and natural language generation capabilities, we propose a simple paradigm for detecting bugs in graph database engines. Compared to existing bug detection methods, this paradigm does not require extensive

**Figure 1: An example of labeled property graph**

prior knowledge but can automatically detect bugs in graph database engines using different graph query languages, demonstrating extremely high applicability. Additionally, benefiting from the diverse and complex queries generated by this paradigm, it can detect wrong-result bugs in the latest versions of graph database engines that prove challenging for existing bug detection methods to identify. Specifically, we start by randomly generating a property graph, then input the graph data, query generation instructions, and query generation constraints into ChatGPT to generate queries that meet these conditions. Finally, we run the queries on different graph database engines and detect bugs based on the discrepancies in query results.

This paper has made the following contributions:

- We designed a prompt template that helps ChatGPT generate a high proportion of non-empty result queries and enhances the diversity of generated queries, improving the efficiency of detecting bugs.
- We proposed a simple paradigm for detecting bugs based on ChatGPT, which does not require a significant amount of prior knowledge and can be applied to graph database engines using different graph query languages.
- We applied this paradigm to graph database engines using Cypher and Gremlin languages, effectively detecting 7 and 3 wrong-result bugs, respectively, demonstrating the effectiveness and applicability of the proposed paradigm.

## 2 A SIMPLE PARADIGM FOR DETECTING GRAPH DATABASE ENGINE BUGS

In this section, we introduced a simple paradigm, DGDB, that utilizes a large language model for automatically detecting bugs in graph database engines. As depicted in Figure 2, this paradigm mainly consists of three stages. In the first stage, graph data used for testing bugs is generated based on randomly generated graph schema. In the second stage, the graph data, instructions for generating queries, and constraints for query generation are input into a large language model (such as ChatGPT) to generate queries for testing bugs. In the third stage, after inputting generated queries into different graph database engines, detecting the presence of bugs is based on the comparison of consistency of the query results. We choose the most popular ChatGPT as the large language model used in the paradigm.

### 2.1 GRAPH DATA GENERATION

Graph databases utilize various graph models to store graph data, with the most commonly used being the labeled property graph model[46] and the Resource Description Framework (RDF) graph model[1].

In the Labeled Property Graph Model, both nodes and edges in the graph are endowed with labels and properties. Labels represent the type of nodes or edges, while properties describe the characteristics of nodes or edges in the form of key-value pairs. As shown in Figure 1, this labeled property graph contains two nodes labeled as "user", two nodes labeled as "product", three edges labeled as "purchase", and one edge labeled as "make friends with". The numbers to the left of the labels represent IDs. Nodes labeled as "user" have a "name" property, while nodes labeled as "product" have "name" and "price" properties. Edges labeled as "purchase" have a "date" property representing the purchase date, and edges labeled as "make friends with" have a "since" property indicating the date when the friendship with the user was established. The Labeled Property Graph Model provides a flexible and intuitive alternative for data modeling, precisely capturing and expressing the intricate relationships between nodes.

The RDF graph model utilizes triples to describe relationships between resources, with each triple consisting of a subject, predicate, and object. The subject represents a resource, typically depicted as a node or edge in graph data. The object signifies information or values related to the subject, often represented as nodes, edges, or actual values in graph data. The predicate denotes the relationship between the subject and object. For example, the RDF triple describing node 1 in Figure 1 can be represented as: (subject: node 1, predicate: type, object: User), (subject: node 1, predicate: name, object: Alice). The RDF graph model benefits from its triple structure, providing stronger semantic expressiveness, and is widely applied in the field of knowledge graphs. However, when dealing with large-scale graph data, using triples requires more storage space and complex indexing structures, resulting in decreased query efficiency. Therefore, we opt for the labeled property graph model to generate a graph database for testing bugs. Specifically, we randomly generate the graph schema of the labeled property graph and, based on this schema, randomly generate graph data in the graph database engine.

---

#### Algorithm 1 Graph data generation

---

**Input:** *NodeLabelSet*, *EdgeLabelSet*, *PropertySet*, the number of nodes  $M$ , the number of edges  $N$ , the property key-value pair generation distribution  $p$ .

```

1:  $v\_set \leftarrow \emptyset$ ,  $e\_set \leftarrow \emptyset$ 
2: for  $i = 1, 2, \dots, M$  do
3:    $v \leftarrow$  create a new node.
4:    $v.label \leftarrow \text{random.choice}(VerLabelSet)$ 
5:    $v.properties \leftarrow$  generate the properties according to  $p$  and  $PropertySet$ .
6:    $v\_set \leftarrow v\_set \cup v$ 
7: end for
8: for  $j = 1, 2, \dots, N$  do
9:    $v_1 \leftarrow \text{random.choice}(v\_set)$ .
10:   $v_2 \leftarrow \text{random.choice}(v\_set - v_1)$ 
11:   $e \leftarrow$  create a new edge with  $v_1$  as the incoming node and  $v_2$  as the outgoing node.
12:   $e.label \leftarrow \text{random.choice}(EdgeLabelSet)$ .
13:   $e.properties \leftarrow$  generate the properties according to  $p$  and  $PropertySet$ .
14:   $e\_set \leftarrow e\_set \cup e$ .
15: end for
```

---

**Graph schema generation.** The graph schemas generated in the labeled property graph should include node label set *NodeLabelSet*, edge label set *EdgeLabelSet*, and property set *PropertySet*, where *NodeTypeSet*, *EdgeLabelSet*, and *PropertySet* represent the sets of node labels, edge labels, and properties of nodes and edges that appear in the graph, respectively. In our experiment, we let  $NodeLabelSet = \{nt_0, nt_1, nt_2, nt_3\}$ ,  $EdgeLabelSet = \{et_0, et_1, et_2, et_3\}$ ,  $PropertySet = \{name, p_0, p_1, \dots, p_9\}$ . In order to ensure diversity in graph data within the graph database engine, we use *name* within *PropertySet* to represent the unique name of nodes or edges, while the remaining property keys correspond to different types of property values, including *Integer*, *Float*, *String*, and *Boolean*. Additionally, a node or edge can simultaneously contain multiple different property key-value pairs.

**Graph data generation.** Based on the above graph schema, we generate graph data stored in the graph database engine. The algorithm for generating graph data is shown as Algorithm 1 and consists of two parts. In the first part (Lines 1-7), for any generated node  $v$ , we first randomly select any label from *NodeLabelSet* as the label for that node. Then, based on the  $p$  distribution, we select single or multiple properties from *PropertySet* as the properties for that node and generate property values of consistent types randomly. In the second part (Lines 8-15), we first randomly select two different nodes from  $v\_set$  as the in-node and out-node of an edge. We then randomly select any label from *EdgeLabelSet* as the label for that edge. Similarly, based on the  $p$  distribution, we select single or multiple properties from *PropertySet* as the properties for that edge and generate property values of consistent types randomly. We set the  $p$  distribution to  $[0.8, 0.1, 0.05, 0.05]$ , and the corresponding numbers of property key-value pairs for nodes or edges are  $[1, 2, 3, 4]$ . Note that the *name* property key-value pairs are not included in this count.

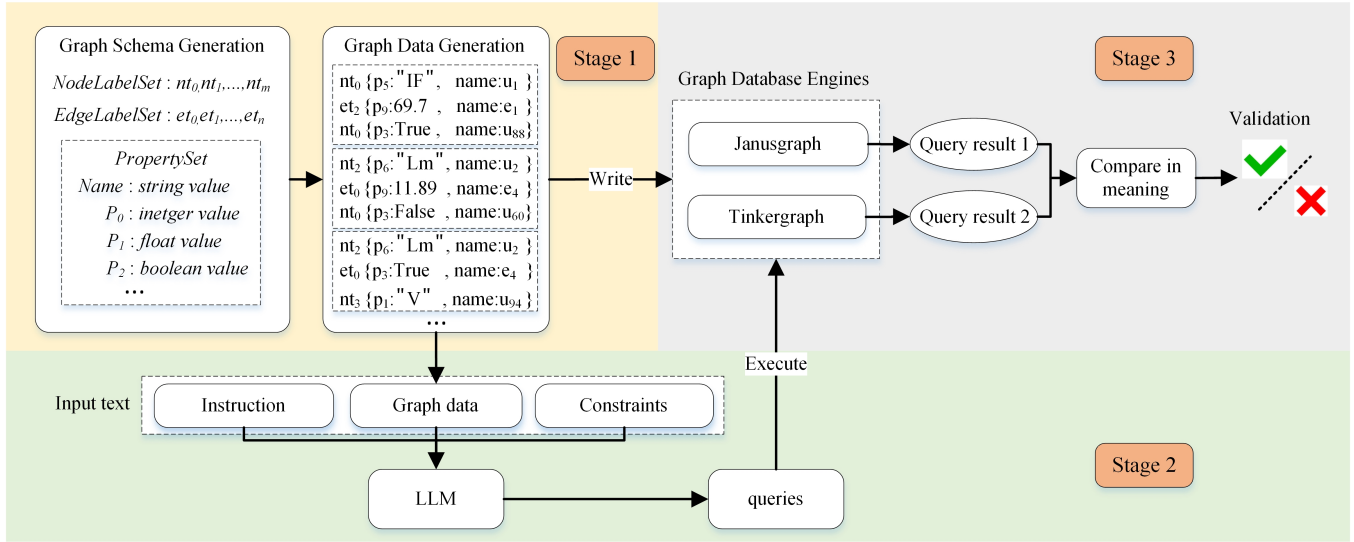


Figure 2: Overview of DGDB

## 2.2 QUERY GENERATION

Traditional methods for generating queries[20, 24, 50] often require designing complex algorithms and an extensive prior knowledge to achieve a high non-empty-result query ratio. In contrast, ChatGPT simplifies this process. With its demonstrated ability in zero-shot in-context learning across various text generation tasks, ChatGPT can achieve high non-empty-result query ratio by simply receiving the appropriate instructions alone[6, 32, 48].

**Instruction:** Your task is to generate queries in the Graph database according to the nodes and relationships in the mentioned graph. The edges in the graph are represented as (node type attribute key value pairs)-(relationship type attribute key value pairs)->(node type attribute key value pairs). For example, (nt0 p5: "IF",name: "u1")-(et2 name: "e1",p9: 69.7)->(nt0 p3: true,name: "u88") indicates that there exists a directed edge of type et2 from a node named "u1" to a node named "u88".

The following is the graph data in the Graph database engine, which contains 100 nodes and 200 edges:

(nt0 p5: "IF",name: "u1")-(et2 name: "e1",p9: 69.7)->(nt0 p3: true,name: "u88").

(nt0 p5: "IF",name: "u1")-(et0 p6: "7W",name: "e0")->(nt0 p1: "a",name: "u21").

...

(nt1 p3: false,name: "u99")-(et0 p0: 69.37,name: "e199")->(nt0 p0: 64.66,p1: "gL",name: "u30",p8: 51.93).

**Query:** Based on the instruction and graph data, Please generate generate a specific number of queries with the different operators (eg. some operators used in a specific graph query language,) and meet the following conditions: (Some constraints.)

To utilize ChatGPT for generating meaningful queries, we start by giving ChatGPT a specific task through Instructions. Next, we explain the specific constructions of graph data to it using examples. This step is crucial to offer ChatGPT an unbiased perception of the graph data. Then we input the graph data into ChatGPT, based on which allows it to understand the complex structure and semantic relationship of graph data. Finally, we ask ChatGPT to generate corresponding query statements. To enhance the diversity of queries and improve the ratio of non-empty -result query, we set various constraints in the query formulation process.

**Listing 1: An example of a query that is consistent in meaning but not entirely consistent in return results.**

1. **Query:** **MATCH** (n) **WHERE** n.name = 'u16' **RETURN** n;
2. **Neo4j:** [Node('nt0', name='u16', p4=False, p7=True, p9=41.96)].
3. **Agensgraph:** ('nt0[4.4]{'p4': false, 'p7': true, 'p9': 41.96, 'name': 'u16'}).

## 2.3 BUG DETECTION

The differential testing method[30], is widely used to detect bugs and errors in software programs due to its ability to discover new problems, facilitate continuous improvements, and be applied universally. Therefore, we use the differential testing method to detect bugs in the graph database engine. Specifically, our process begins by selecting graph database queries generated by ChatGPT as initial test samples. Subsequently, we refine selected queries by filtering out those that either yield random query results or alter the original graph data to form our final set of test samples. Following that, we execute the same test samples on different graph database engines. Ultimately, we compare the outputs of different graph database engines. Since different graph database engines handle nodes or edges differently, requiring completely consistent query results may lead to discrepancies that not necessary reflect the existence of bugs. As shown in listing 1, the query results are not completely consistent,

but they all represent node 'u16'. Therefore, in order to improve the efficiency of bug detection, we believe that if the query results of different database engines are consistent in meaning, then the test sample passes; otherwise, a bug may have occurred.

### 3 DGDB EXAMPLES

In this section, we provide two examples demonstrating how to use DGDB to detect bugs in graph database engines: DGDB-Cypher for detecting bugs in graph database engines using Cypher query language and DGDB-Gremlin for detecting bugs in graph database engines using Gremlin query language.

#### 3.1 DGDB-CYPHER

For the bug detection in graph database engines based on the Cypher query language, we selected Neo4j and AgensGraph as the target databases. Neo4j[22] is a high-performance open-source graph database engine designed for storing, querying, and analyzing graph data. It is widely used due to its efficient execution of complex graph queries and the capability to handle large-scale graph data. AgensGraph[21] is a multi-model graph database engine known for its flexibility in handling different types of data, including graph data and relational data, making it widely utilized.

DGDB-Cypher first constructs corresponding graph data in Neo4j and AgensGraph, ensuring that the node types, node properties, edge types and edge properties are identical. The constructed graph data is then inserted into the instruction, fed into ChatGPT, and used to generate Cypher queries. The text inputted into ChatGPT is as follows:

**Instruction:** Your task is to generate queries in the Graph database according to the nodes and relationships in the mentioned graph. The edges in the graph are represented as | (: node type attribute key value pairs) | (: relationship type attribute key value pairs) | (: node type attribute key value pairs) |. For example, | (: nt8 p6: false, name: "u97") | (: et11 p8: true) | (: nt6 p17: false, name: "u33") | Indicates that there exists a directed edge of type et11 from a node named "u97" to a node named "u33".

The following is the graph data in the Graph database engine, which contains 100 nodes and 200 edges:

```
|(:nt0 p5: "IF",name: "u1")|(:et2 name: "e1",p9: 69.7)|(:nt0 p3: true,name: "u88")|
|(:nt0 p5: "IF",name: "u1")|(:et0 p6: "7W",name: "e0")|(:nt0 p1: "a",name: "u21")|
|(:nt2 p6: "Lm",name: "u2")|(:et0 p1: "Glo",name: "e4",p9: 11.89)|(:nt0 p3: false,name: "u60")|
...
|(:nt1 p3: false,name: "u99")|(:et0 p0: 69.37,name: "e199")|(:nt0 p0: 64.66,p1: "gL",name: "u30",p8: 51.93)|
```

Based on the instruction and graph database, Please generate twenty cypher queries with the different operators(eg. MATCH, OPTIONAL MATCH, WHERE, Aggregation, FOREACH, RETURN, ORDER BY, WITH, UNWIND, UNION, UNION ALL, collect, predicate, coalesce, length, type, keys, labels,

startNode, endNode, nodes, relationships, reduce, shortestPath) and meet the following conditions:

1. Please make sure the queried data is the node or link mentioned earlier.
2. Please ensure that the values in the attribute key value pairs of the constraint exist.
3. If you want to generate a query with relationships, please pay attention to the direction of the query relationships in the generated query statement.
4. Please ensure that the generated queries use different keywords as much as possible.
5. Please ensure that the generated query statement will not change the data of the Graph database(eg. Do not use the create operators).

In the above text, the modified parts are highlighted in gray. It can be observed that we mainly made changes to the types, quantity, operators and constraints of generated queries.

- **Types and quantity:** We stipulate the generation of Cypher queries; and due to the token limit of ChatGPT, we set the quantity of generated Cypher queries to 20.
- **Operators:** We hope that ChatGPT uses different operators when generating queries to avoid situations where the generated queries are highly similar, as shown in Listing 2, and to improve detection efficiency.
- **Constraints:** We have set a total of five constraints. The first two points indicate that we hope ChatGPT can generate as many non-empty-result queries as possible. The third constraint is to ensure that ChatGPT considers the direction of edges when generating query statements because we found that without this constraint, ChatGPT tends to ignore the direction of arrows in most generated query statements. The fourth constraint is to ensure that ChatGPT can generate more complex query statements. The fifth constraint is to ensure consistency in graph data across different graph database engines.

**Listing 2:** An example of generating highly similar queries, where only the types of nodes or edges that they belong to are changed in the queries.

```
1. MATCH (n:nt1)-[:et1]->(m:nt3) RETURN n, m;
2. MATCH (n:nt1)-[:et1]->(m:nt2) RETURN n, m;
```

The above modifications were mostly made to generate high-quality Cypher query statements, improving the efficiency of detecting bugs. We iterated a total of 200 times, generating 4000 query statements. We then filtered out queries containing operators that produce random results, such as queries including the skip or limit keywords. The remaining query statements were executed on Neo4j and AgensGraph graph database engines, and the query results were compared. Due to discrepancies in internal implementation, data storage structure, and performance optimization among different graph database engines, there may be variations in query results. As shown in Listing 3, the returned results consist of lists containing nodes such as u1, u3, but their presentation may differ. Compared to using a mapping table for data transformation, which



introduces some additional computational and storage costs[50], our method utilizes regular expressions[27] to directly extract information about corresponding nodes from query results, enhancing the efficiency of bug detection. Finally, we compare this information, and if discrepancies arise, it indicates that the graph database engine may have a bug in executing this query statement.

**Listing 3: An example of using regular expressions to extract information from cypher query results.**

```
1. Query: MATCH (n:nt0) WITH collect(n) as n RETURN DISTINCT n;
2. Neo4j: [Node('nt0', name='u1', p5='IF'), Node('nt0', name='u3', p4=True, p6='9')...] - regular expression -> [{name: 'u1', p5: 'IF'}, {name: 'u3', p4: True, p6: '9'}...].
3. Agensgraph: [{id: '4.1', tid: '(0,1)', properties: {p5: 'IF', name: 'u1'}}, {id: '4.2', tid: '(0,2)', properties: {p4: True, p6: '9', name: 'u3'}}...] - regular expression -> [{name: 'u1', p5: 'IF'}, {name: 'u3', p4: True, p6: '9'}...].
```

### 3.2 DGDB-GREMLIN

For the bug detection in graph database engines based on the Gremlin query language, we selected JanusGraph and TinkerGraph as the target database engines. JanusGraph[2] is a high-performance distributed graph database and widely used for handling large-scale graph data and complex graph queries. TinkerGraph[15] is a lightweight database in the TinkerPop graph computing framework, widely applied in prototype development for small-scale projects due to its superior read and write performance storing data in memory.

DGDB-gremlin first constructs graph data in JanusGraph and TinkerGraph separately, ensuring consistency in the stored graph data between the two graph database engines. The constructed graph data is then converted into the context, which is then fed into ChatGPT to generate Gremlin query statements. The specific text input to ChatGPT is as follows:

**Instruction: Your task is to generate queries in the Graph database according to the nodes and relationships in the mentioned graph. The edges in the graph are represented as (node type attribute key value pairs)-(relationship type attribute key value pairs)-(node type attribute key value pairs). For example, (nt0 p5: "IF",name: "u1")-(et2 name: "e1",p9: 69.7]->(nt0 p3: true,name: "u88") Indicates that there exists a directed edge of type et2 from a node named "u1" to a node named "u88".**

The following is the graph data in the Graph database engine, which contains 100 nodes and 200 edges:

```
(nt0 {p5: "IF",name: "u1"})-(et2 {name: "e1",p9: 69.7}]->(nt0 {p3: true,name: "u88"})
...
(nt1 {p3: false,name: "u99"})-(et0 {p0: 69.37,name: "e199"})->:(nt0 {p0: 64.66,p1: "gL",name: "u30",p8: 51.93}).
```

Based on the instruction and graph database, Please generate twenty gremlin queries with the different operators(eg.

hasLabel(), hasId(), has(), hasNot(), values(), label(), id(), properties(), values(), valueMap(), select(), dedup(), local(), order(). by(), where(),filter(),match(),eq(), neq(), gt(), gte(), inside(), outside(), group().by(), groupCount().by(), in(), out(), inE(), outE(), inV(), outV(), both(), path(), repeat().until(), sum(), max(), min(), mean(),contains(), choose(), union(), fold()) and meet the following conditions:

1. Please make sure the queried data is the node or link mentioned earlier.
2. Please ensure that the values in the attribute key value pairs of the constraint exist.
3. If you want to generate a query with relationships, please pay attention to the direction of the query relationships in the generated query statement.
4. Please ensure that the generated queries use different keywords as much as possible.
5. Please ensure that the generated query statement will not change the data of the Graph database(eg. Do not use the addV() or addE() operators).

Similar to the Instruction for generating Cypher query statements, we made modifications to the types, quantity, operators, and constraints for generating Gremlin query statements. The details are as follows:

- **Types and quantity:** We set the total 20 types of generated query statements to Gremlin.
- **Operators:** We select a large number of commonly used operators in Gremlin to ensure the diversity of the generated Gremlin query statements.
- **Constraints:** The constraints for generating Gremlin query statements are generally similar to those for generating Cypher query statements. Both aim for ensuring complex structures and high non-empty-result ratio for queries generated by ChatGPT, thereby improving the efficiency of detecting graph database engine bugs.

Like DGDB-CYPHER, we use the differential testing method to detect bugs for Gremlin-based graph database engines. Notably, both JanusGraph and TinkerGraph return results as node ID for node querying requests. Furthermore, since JanusGraph automatically generate node ID within the graph database for maximizing data consistency, it is challenging for us to directly determine whether the query results are identical solely based on node IDs. Our solution is to compare the query results by obtaining the correspondent properties to the node IDs, as exhibited in Listing 4.

**Listing 4: An example of comparing node IDs in gremlin query results.**

```
1. Query: g.V().hasLabel('nt0');
2. Janusgraph: [v[16416], v[24608], v[32800],...].
3. Tinkergraph: [v[3], v[9], v[278],...].
4. Query: g.V().hasLabel('nt0').valueMap()
5. Janusgraph: [{name: 'u16', p9: [41.96], p4: [False], p7: [True]}, {name: 'u25', p5: ['UR']},...]
6. Tinkergraph: [{p5: ['IF'], name: 'u1'}, {p4: [True], p6: ['9'], name: 'u3'}],...]
```

## 4 EVALUATIONS

To demonstrate the effectiveness of the proposed DGDB paradigm, we compare DGDB-cypher and DGDB-gremlin respectively with the GDSmith and Grand methods, and address the following research questions:

- (RQ1) How does the quality of queries generated by ChatGPT stand in comparison to those produced by the baseline model?
- (RQ2) Can the graph database engine bug detection methods proposed based on the DGDB paradigm detect real-world bugs in popular graph database engines?

### 4.1 EVALUATION SETUP

**4.1.1 Subjects.** We use listed graph database engines in Table 2 for detecting bugs, and select the latest versions in all cases since new versions of graph database engines may have addressed bugs presented in previous versions while detecting bugs in previous versions might reveal issues that have already been fixed. Moreover, detecting bugs in new versions that are more likely to be persisted in old versions makes our tasks more meaningful.

**Table 2: The graph database engines selected for testing**

GDBMS	Version	Release date	Supported languages
Neo4j	4.4.26	2023.9.20	Cypher
Agensgraph	2.13.0	2022.10.13	Cypher
Janusgraph	1.0.0	2023.10.21	Gremlin
Tinkergraph	3.7.0	2023.7.31	Gremlin

**4.1.2 Baseline.** To demonstrate the superiority of the proposed paradigm, we compare two methods for graph database engine bug detection, as follows:

- GDSmith[20]: GDSmith is a bug detection method for Cypher-based graph database engines. It first guides the generation of semantically valid Cypher queries based on property graphs. Subsequently, the generated queries are executed on each graph database engine and identify potential bugs based on discrepancies of the outputs.
- Grand[50]: Grand is a bug detection method for Gremlin-based graph database engines. It first constructs a traversal model based on Gremlin API, based on which to generate effective Gremlin queries. Subsequently, the generated queries are executed on different graph database engine, and bugs are detected based on discrepancies between the query results.

**4.1.3 Implementation details.** In our research, we implement DGDB-cypher and DGDB-gremlin through Python 3.8. We use the Py2neo package to connect to the Neo4j for operations, the Psycogp2 package to connect to the AgensGraph for operations, and the Gremlin-python package to connect to JanusGraph and TinkerGraph for operations. The version of the ChatGPT model is gpt-3.5-turbo-16k-0613. The versions of each package are as shown in Table 3. The numbers of generated nodes, edges, node types, edge types, and property types are set to 100, 200, 4, 4, and 11, respectively. We set the number of rounds for both DGDB-cypher and DGDB-gremlin to 200, generating 4000 query statements respectively.

**Table 3: The version of each package we used**

Package	Version
Py2neo	2021.2.3
Psycogp2	2.9.7
Gremlin-python	3.2.6
Openai	0.27.10

### 4.2 PERFORMANCE COMPARISON (RQ1)

Grand[50], as the first method proposed for detecting graph database engine bugs, utilizes the differential testing method to detect bugs in graph database engines using the gremlin query language. The method ran for 2400 seconds, generating a total of 15,000 queries. It successfully detects a total of 21 bugs across six graph database engines, with most of the bugs already being fixed. However, this method often detects a significant number of discrepancies, as evidenced by the 709 discrepancies found among 15,000 generated queries in the original study and the 615 discrepancies observed in 10,000 generated queries when Grand is used to detect bugs in Janusgraph, TinkerGraph and HugeGraph, as mentioned in [24]. These discrepancies are largely attributed to variations in handling exceptions by different graph database engines rather than detecting actual bugs in graph database engines<sup>1</sup>. GDSmith[20] is the first graph database bug detection method based on the Cypher query language. It ran for 12 hours in neo4j, redisgraph, and memgraph, discovering a total of 11,275 discrepancies and identifying 28 bugs from these discrepancies, with the majority of them already fixed. It can be observed that both methods mentioned above have discovered a significant number of discrepancies. However, a large number of discovered discrepancies may merely indicate the presence of numerous duplicated bugs or results from different internal implementations of various graph database engines, leading to different outputs. Furthermore, analyzing each discrepancy can result into substantial time cost, and naturally compromise the efficiency of bug detection. Therefore, we believe that the quantity of discrepancies should not be used as a benchmark for the performance of graph database engine bug detection. Moreover, since most of the bugs detected by the two methods mentioned above have already been fixed through the version updates, we cannot simply compare the performance of graph database engine bug detection methods based solely on the number of detected bugs. Our methods DGDB-cypher and DGDB-gremlin each ran for about 4200 seconds, generated 4000 queries, and collectively detected 10 unfixed bugs in the four latest versions of popular graph database engines, which exclude the solved bugs, and therefore, offer more convincing evidences for demonstrating the effectiveness of our methods.

DGDB-cypher, DGDB-gremlin, Grand, and GDSmith all use the differential testing method to detect graph database engine bugs, where the quality of generated queries determines the performance of differential testing methods. Therefore, we analyzed the quality of queries generated by these four methods. We believe that high-quality queries should possess two essential criteria: it should be

<sup>1</sup><https://github.com/choeoe/Grand/issues/1>

diversified and have a high non-empty-result query ratio. Specifically, diverse queries can comprehensively detect potential graph database bugs introduced by different operators, and high non-empty-result queries can significantly improve the efficiency of bug detection.

**Listing 5: An example of a query on search paths.**

```
1. MATCH p=shortestPath((n1)-[:et0*]-(n2:nt0)) WHERE n1 <>
n2 RETURN n1, n2;
```

- **Diversity.** The Grand method constructs a traversal model to generate queries, including three types of operations: filter, predicate, and aggregate. However, due to the limitations of this traversal model, the generated queries may not include certain operators such as *repeat()*, *until()*, *path()*, *fold()*, etc., which hinder the detection of bugs associated with these operators. The GDSmith method firstly constructs a Cypher skeleton, then generates subgraphs that match the property graph to complete the pattern of the Cypher skeleton. Finally, based on static query analysis, the conditional expressions of the Cypher skeleton are completed to obtain the generated queries. However, this method only queries elements in the matching subgraphs and cannot generate statements for searching paths, as shown in Listing 5. It can be seen that, due to limitations in the methods of generating queries, Grand and GDSmith cannot generate more comprehensive queries. If one wants to increase the diversity of generated queries, extensive expert knowledge is required to make complex modifications to existing methods, resulting in excessive manual cost. Comparably, DGDB-cypher and DGDB-gremlin exhibit advantages in utilizing more flexible and convenient operators in the form of textual input to ChatGPT. For example, in DGDB-gremlin, "Please generate twenty gremlin queries with different types of methods (e.g., *hasLabel()*, *hasId()*, *has()*, *hasNot()*, *values()*, *[operators]*)", we only need to replace the *[operators]* with the operators we want to use in the generated queries to ensure that the generated queries include that operator, creating diverse queries for a more comprehensive detection of bugs in the graph database engine.
- **Non-empty-result query ratio.** As shown in Table 4, we compared the non-empty-result query ratios generated by DGDB-cypher, DGDB-gremlin, Grand, and GDSmith. It can be observed that the non-empty-result query ratio of queries generated by DGDB-cypher and DGDB-gremlin is higher than those by GDSmith and Grand. This indicates that ChatGPT can understand graph data well and generate high-quality queries based on constraints. Additionally, the non-empty-result query ratio of Grand is much lower than GDSmith. This is because Grand is a random differential testing method, while GDSmith improves the non-empty-result query ratio by using graph-guided pattern generation and data-guided condition generation.

In summary, compared to existing methods for detecting bugs in graph database engines, utilizing the DGDB paradigm allows for a more comprehensive and efficient detection of bugs in graph database engines.

**Table 4: The non-empty-result query ratio of DGDB and two baselines**

Approach	Language	Non-empty-result Query Ratio
DGDB-cypher	Cypher	79.06%
GDSmith	Cypher	73.66%
DGDB-gremlin	Gremlin	80.33%
Grand	Gremlin	40.30%

### 4.3 DETECTED WRONG-RESULT BUGS (RQ2)

Bugs detected using differential testing mainly fall into two categories: crash bugs[17] and wrong-result bugs[20]. Crash bugs occur when users run certain commands and do not receive the expected results; instead, it causes the graph database engine to crash. Since users cannot obtain the expected results, they become aware of the bug. Grand and GDSmith detect many bugs of this type. Wrong-result bugs are referred to the incorrect results obtained based on users' running commands. These erroneous results often resemble with correct ones, making it challenging for users to identify manually and, consequently, leading to greater risks compared to the crash bugs. Therefore, our primary focus is on detecting wrong-result bugs.

**Table 5: Bugs that we found in the tested graph database engines**

Approach	Graph database engine	Number of wrong-result bugs detected
DGDB-cypher	Neo4j	2
DGDB-cypher	Agensgraph	5
DGDB-gremlin	Janusgraph	3
DGDB-gremlin	Tinkergraph	0

Table 5 indicates the number of bugs detected using DGDB-cypher and DGDB-gremlin, and we present the specific bugs in the subsequent content. Notably, we choose to maximize the clarity for our readers by presenting the succinct yet illustrative examples in the following content instead of the original queries that are implemented to discover the bugs since the original queries are typically too complex to be faithfully showcased in this paper.

**Listing 6: Cypher queries that trigger a wrong-result bug in Neo4j 4.4.26.**

```
1. Query 1: MATCH (n:nt3) RETURN count(n), avg(n.p8);
2. Neo4j: [3, 25.786666666666667] ✗
3. Agensgraph: (24, 25.786666666666665) ✓
4. Query 2: MATCH (n:nt3) RETURN count(n);
5. Neo4j: [26] ✓
6. Agensgraph: (26,) ✓
```

As shown in listing 6, query 1 returns the number of nodes of type "nt3" and the mean value of the "p8" property value for nodes of that type. It can be observed that Neo4j and AgensGraph



return different node counts. However, the node count returned by query 2 is the same. Therefore, we believe the reason for the bug is that Neo4j, when simultaneously returning the node count and node property values, first considers whether the "p8" property value exists in the properties of nodes of type "nt3". If it exists, it is counted; otherwise, it is not counted, leading to the bug.

**Listing 7: Cypher queries that trigger a wrong-result bug in Neo4j 4.4.26.**

```
1. Query: MATCH (n:nt1)-[r]-() RETURN n.name AS Name,
sum(r.p8) AS TotalP8;
2. Neo4j: [['u5', 0], ['u6', 17.01], ['u7', 17.02],...,['u22', 84.84]] ✗
3. Agensgraph: [(('u5', None), ('u6', 17.01), ('u7', 17.02),...,('u22',
84.84))] ✓
```

As shown in listing 7, this query returns the names of nodes of type "nt1" and the sum of the "p8" property values for their respective edge sets. It can be observed that there is a difference between the sum of property values returned by Neo4j and AgensGraph. When AgensGraph returns with the "none" result, it indicates that the "p8" property does not exist in the edge set of the node. However, when Neo4j returns its result as "0", it might lead to the misconception that the sum of "p8" property values in the edge set of the node is 0, suggesting that "p8" exists in the edge set of the node. We believe this bug may be due to Neo4j's inadequate handling of exceptional values.

**Listing 8: Cypher queries that trigger a wrong-result bug in Agensgraph 2.13.0.**

```
1. Query 1: MATCH (n)-[r]->() UNWIND n.p6 AS values, RETURN values;
2. Neo4j: [['Lm'], ['GOvy'], ['5Yz'],...,['Rk']] ✓
3. Agensgraph: [] ✗
4. Query 2: MATCH (n)-[r]->() RETURN n.p6;
5. Neo4j: [[None], ['Lm'], ['GOvy'], [None],...,['Rk']] ✓
6. Agensgraph: [(None), ('Lm'), ('GOvy'), (None),..., ('Rk')] ✓
7. Query 3: MATCH (n)-[r]->() WITH n.p6 AS values, RETURN values;
8. Neo4j: [[None], ['Lm'], ['GOvy'], [None],...,['Rk']] ✓
9. Agensgraph: [(None), ('Lm'), ('GOvy'), (None),..., ('Rk')] ✓
```

As shown in listing 8, query 1 unfolds the "p6" property values of nodes with outgoing relationships into separated rows. It can be observed that neo4j returns the correct results, while Agensgraph returns an empty set. Query 2 aims for checking if the data retrieval functions properly by returning the "p6" property values of nodes with outgoing relationships, and the consistent results required from neo4j and Agensgraph confirm them being functional. Query 3 replaces the "unwind" keyword with "with" to check if other operators are causing the issue, and the results from neo4j and Agensgraph are also identical. Therefore, we suspect that there may be a bug in Agensgraph when using the "unwind" keyword to unfold property values.

**Listing 9: Cypher queries that trigger a wrong-result bug in Agensgraph 2.13.0.**

```
1. Query 1: MATCH (n:nt3 {p5: 'Ce'})-[:et3]->(m) RETURN n,
COLLECT(m);
2. Neo4j: [Node('nt3', name='u4', p5='Ce'), [Node('nt1', name =
'u85', p3=True), Node('nt2', name='u84', p9=44.79)]] ✓
3. Agensgraph: [(('nt3[3.2]{"p5": "Ce", "name": "u4"}', [{'id': '5.20',
'tid': None, 'properties': None}], {'id': '6.24', 'tid': None, 'properties':
None}))] ✗
4. Query 2: MATCH (n:nt3 {p5: 'Ce'})-[:et3]->(m) RETURN n,
m;
5. Neo4j: [Node('nt3', name='u4', p5='Ce'), Node('nt1', name =
'u85', p3=True), [Node('nt3', name='u4', p5='Ce'), Node('nt2',
name='u84', p9=44.79)]] ✓
6. Agensgraph: (('nt3[3.2]{"p5": "Ce", "name": "u4"}', 'nt2[5.20]{"p9":
44.79, "name": "u84"}'), ('nt3[3.2]{"p5": "Ce", "name": "u4"}',
'nt1[6.24]{"p3": true, "name": "u85"}') ✓
```

As shown in listing 9, Query 1 first retrieves nodes with the label "nt3" and the property "p5" value 'Ce', then obtains nodes reached along the "et3" relationship from these nodes, and finally aggregates them into a list. Neo4j returns the correct results, while Agensgraph fails to retrieve the node's key-value pairs. Query 2, after removing the "collect" keyword, yields consistent results in both neo4j and Agensgraph. This bug is similar to the one in listing 8, so we suspect that Agensgraph may have a bug when using the "collect" keyword to gather nodes into a list.

**Listing 10: Cypher queries that trigger a wrong-result bug in Agensgraph 2.13.0.**

```
1. Query 1: MATCH (n) WHERE n.p2 > 50 RETURN n.name;
2. Neo4j: [] ✓
3. Agensgraph: [(('u19'), ('u56'), ('u96'), ('u33'), ('u44'), ('u47'),
('u86'), ('u17'), ('u37'), ('u91'))] ✗
4. Query 2: MATCH (n) RETURN n.name, n.p2;
5. Neo4j: [['u17', False], ['u19', True],..., ['u99', None]] ✓
6. Agensgraph: [(('u17', False), ('u19', True)),..., ('u99', None))] ✓
```

As shown in listing 10, query 1 returns the names of all nodes with the property "p2" greater than 50. Neo4j returns an empty set, while Agensgraph returns the names of some nodes. Query 2 returns the names and "p2" property values of all nodes. We can see that the type of the "p2" property value is a boolean constant, and it cannot be compared with an integer constant. However, Agensgraph still returns results. Therefore, we suspect that Agensgraph may have a bug in setting the priority of boolean constant higher than integer constants when comparing values of different types.

As shown in listing 11, query 1 aims to return the count of all nodes going out through the "et0" relationship and returning through the "et3" relationship. Neo4j returns 28, while Agensgraph strangely returns a very large value, 2695, because the total number of nodes in the graph is only 100. Query 2 returns the count of nodes going out through the "et0" relationship, and query 3 returns the

count of nodes returning through the "et3" relationship. The results of these two queries on both graph database engines are consistent, but we found that the product of the results of queries 2 and 3 equals the result of query 1. From this, we can infer that Agensgraph, when using the above pattern to match multiple relationships, does not consider these relationships simultaneously but matches them separately. Finally, it returns the results in a manner similar to a Cartesian product[19], leading to a bug.

**Listing 11: Cypher queries that trigger a wrong-result bug in Agensgraph 2.13.0.**

1. Query 1: <b>MATCH</b> (n)-[:et0]->(), ()-[:et3]->(n) <b>RETURN count</b> (n);	
2. Neo4j: [28] ✓	Agensgraph: (2695,) ✗
3. Query 2: <b>MATCH</b> (n)-[:et0]->() <b>RETURN count</b> (n);	
4. Neo4j: [55] ✓	Agensgraph: (55,) ✓
5. Query 3: <b>MATCH</b> ()-[:et3]->(n) <b>RETURN count</b> (n);	
6. Neo4j: [49] ✓	Agensgraph: (49,) ✓

**Listing 12: Cypher queries that trigger a wrong-result bug in Agensgraph 2.13.0.**

1. Query 1: <b>MATCH</b> (n1)-[r]->(n2:nt0) <b>WHERE</b> n1.name = 'u9' <b>RETURN COLLECT(DISTINCT</b> n2.p2) <b>AS</b> distinct_values;	
2. Neo4j: [[False]] ✓	Agensgraph: ([False, None],) ✗

As shown in listing12, this query first matches nodes connected to the 'u9' node with the type 'nt0' and then returns the unique set of p2 property values for these nodes. The result returned by Neo4j is 'False', while Agensgraph returns 'False' and 'None'. This is because in Agensgraph, None indicates the absence, meaning that some nodes in the matched nodes may lack the p2 property, resulting in the presence of none. However, Distinct is meant to return unique elements, and after using distinct, None values should be removed to avoid potential user confusion.

**Listing 13: Gremlin queries that trigger a wrong-result bug in Janusgraph 1.0.0.**

1. Query 1: g.E().has('p2', without('GhR')).count();	
2. Janusgraph: [9] ✗	Tinkergraph: [23] ✓
3. Query 2: g.E().has('p2').count();	
4. Janusgraph: [23] ✓	Tinkergraph: [23] ✓
5. Query 3: g.E().has('p2', without('GhR')).valueMap(); 6. Janusgraph: [{ 'name': 'e16', 'p2': True }, { 'p2': True, 'name': 'e13', 'p9': 55.07 }, ..., { 'name': 'e180', 'p2': True } ] ✗	
7. Tinkergraph: [{ 'p2': False, 'name': 'e5' }, { 'p1': '5k', 'p2': False, 'p4': False, 'name': 'e9' }, { 'p2': True, 'name': 'e16' }, ..., { 'p2': True, 'name': 'e13', 'p9': 55.07 } ] ✓	

As shown in listing 13, query 1 returns the number of edges that have the "p2" property and the "p2" property value does not contain 'GhR'. It can be seen that Janusgraph and tinkergraph return different values. Query 2 returns the number of edges that have the "p2" property, and it is observed that the results of the two

graph database engines are consistent. Therefore, we believe that there is an issue with the use of the without operator. Subsequently, we use query 3 to return the property key-value pairs of edges that have the "p2" attribute, and the "p2" property value does not contain 'GhR'. We found that Janusgraph only returns edges with "p2" attribute value as True. It can be inferred that Janusgraph has a bug when using the without operator to handle property values of different types.

**Listing 14: Gremlin queries that trigger a wrong-result bug in Janusgraph 1.0.0.**

1. result_set = client1.submit("g.addV('nt5').property('p9', 13.85).property('test3', 'pvifo')")	
2. nodes = list(map(lambda v: "label": g.V(v).label().toList(), "properties": g.V(v).valueMap().toList(), g.V()))	
3. for node in nodes:	
4. print("Node Properties:", node["properties"])	
5. Janusgraph_Output: Node Properties: [{ 'p9': [13.85], 'test3': ['pvifo'] }] ✓	
6. Tinkergraph_Output: Node Properties: [{ 'test3': ['pvifo'], 'p9': [{ '@type': 'gx:BigDecimal', '@value': 13.85 } ] }] ✓	
7. g.V().drop().iterate(); g.E().drop().iterate();	
8. result_set = client1.submit("g.addV('nt5').property('p9', 13).property('test3', 25.6)")	
9. nodes = list(map(lambda v: "label": g.V(v).label().toList(), "properties": g.V(v).valueMap().toList(), g.V()))	
10. for node in nodes:	
11. print("Node Properties:", node["properties"])	
12. Janusgraph_Output: Node Properties: [{ 'p9': [13.0], 'test3': ['25.6'] }] ✗	
13. Tinkergraph_Output: Node Properties: [{ 'test3': [{ '@type': 'gx:BigDecimal', '@value': 25.6 } ], 'p9': [13] }] ✓	

As shown in listing 14, we provide a partial code example in Python to create graph data. Lines 1-4 create a node and return its property key-value pairs. Line 7 deletes all nodes and edges from the graph. Lines 8-11 create a node with the same type but different property types and return its property key-value pairs. From the output results in lines 5 and 6, it can be seen that the property values of nodes constructed under different graph database engines are the same. From the result in line 12, it can be seen that Janusgraph creates the 'p9' property value and 'test3' property value of the node with integer and float, but displays them as float and string. Line 13 shows that Tinkergraph displays property values in a type consistent with the input during node creation. Therefore, we infer that even after deleting all nodes and edges from the graph, the property value types of the nodes initially constructed in Janusgraph will still affect the types of subsequently constructed node property value, leading to a bug.

As shown in listing 15, query 1 returns the key-value pairs of nodes with the 'p2' property and 'p2' property value not equal to the string 'false'. The node property key-value pairs returned by Janusgraph are far fewer than those returned by Tinkergraph. Query 2 returns the key-value pairs of nodes with the 'p2' property

and 'p2' property value not equal to the boolean value false, and it can be seen that the results returned by Janusgraph and Tinkergraph are consistent. By comparing the results of query 1 and query 2 in Janusgraph, we can conclude that Janusgraph treats the string 'false' property value as a boolean value false, leading to this bug.

**Listing 15: Gremlin queries that trigger a wrong-result bug in Janusgraph 1.0.0.**

```
1. Query 1: g.V().has('p2', neq('false')).valueMap();
2. Janusgraph: ['p2': [True], 'name': ['u96'], 'p2': [True], 'name': ['u19'], 'p2': [True], 'name': ['u47']] ✗
3. Tinkergraph: ['p2': [False], 'name': ['u91'], 'p2': [True], 'name': ['u96'], ..., 'p2': [True], 'name': ['u47']] ✓
4. Query 2: g.V().has('p2', neq(false)).valueMap();
5. Janusgraph: ['p2': [True], 'name': ['u96'], 'p2': [True], 'name': ['u19'], 'p2': [True], 'name': ['u47']] ✓
6. Tinkergraph: ['p2': [True], 'name': ['u96'], 'p2': [True], 'name': ['u19'], 'p2': [True], 'name': ['u47']] ✓
```

From the above bug analysis, it can be seen that bugs in graph database engines mainly occur in the comparison of different types of property values (e.g., Listing 10, Listing 13), the use of certain operators (e.g., Listing 8, Listing 9), execution order under complex queries (e.g., Listing 6), default settings for null and boolean values (e.g., Listing 7, Listing 12, Listing 15), handling logic under complex pattern matching (e.g., Listing 11), and incomplete execution of some commands (e.g., Listing 14). The above-mentioned bugs have all been submitted to the corresponding communities of the graph database engines on github.

## 5 RELATED WORK

Recently, researchers have proposed various methods for detecting bugs in relational database engines. SQLsmith[3] is a fuzz testing method for detecting bugs in relational database engines. It first generates SQL query randomly according to the Abstract Syntax Tree (AST) generator, then simplifies the generated SQL query through SQL Reducer. Finally, it determines the presence of bugs based on whether the generated SQL query can successfully run on the relational database engines. While this method is effective in detecting crash bugs, it cannot identify wrong-result bugs. PQS[38] selects a target data from randomly generated tables, generates conditional expressions based on the target data, constructs an SQL query with a where or join clause, and determines the presence of bugs by checking if the result is included in the result set. While this method can effectively detect bugs in relational database engines, its applicability is limited across different relational database engines due to variations in SQL syntax and query optimization strategies. NoRec[36] converts the original SQL query into a non-optimized SQL query and compares the results of these two SQL queries for consistency. However, this method cannot detect bugs in SQL queries with subquery structures. TLP[37] transforms randomly generated original SQL queries into three different logical queries based on the true, false, and null ternary logic. It detects bugs in relational database engines by comparing the result sets of the transformed three SQL queries with the original SQL query.

TQS[45] proposes a testing framework capable of detecting logical bugs arising from multi-table join queries. The authors treat the database schema as a graph and use biased random walks to generate SQL queries. Based on biased random walks and graph embedding methods, high-quality SQL queries are generated to detect numerous logical bugs in relational database engines.

In contrast to detecting bugs in relational database engines, the identification of bugs in graph database engines is relatively less discussed. This is due to the significantly earlier development of relational database engines compared to graph database engines. However, significant differences exist in data models, query languages, storage structures, etc., between relational and graph database engines. As a result, bug detection methods designed for relational database engines cannot be directly applied to graph database engines. Moreover, because graph database engines use nodes and edges to represent entities and their relationships, the data model of graph database engines becomes more flexible and complex. For instance, nodes can have an arbitrary number of properties and relationships, increasing the complexity of bug detection in graph database engines. Furthermore, unlike relational database engines that have a universal and standardized query language like SQL[7], each graph database engine has its proprietary query language. For instance, AgensGraph uses Cypher, and JanusGraph uses Gremlin, making it challenging to design a bug detection method that is universally applicable across different graph database engines.

The detection method for bugs in graph database engines primarily employs differential testing[13]. By running identical queries on different graph database engines or different versions of graph database engines and comparing their query results, this type of method can effectively identify bugs in graph database engines based on result inconsistencies. For instance, Grand[50] constructed a traversal model to generate syntactically correct and meaningful Gremlin queries. These queries were then input into various graph database engines based on the Gremlin query language, and the results of queries across different graph database engines were compared. GDSmith[20], guided by property graphs, generates Cypher queries, inputs them into graph database engines based on the Cypher query language, and compares the query results across different graph database engines. In contrast to the aforementioned methods, DGDB, based on the recently popularized large language model ChatGPT, proposes a paradigm for detecting bugs in graph database engines. This paradigm, without requiring extensive expert knowledge, is capable of detecting bugs in graph database engines based on different query languages (including but not limited to Cypher, Gremlin) and can increase the non-empty-result ratio of generated queries, significantly enhancing the efficiency of detecting graph database engine bugs.

## 6 CONCLUSION

In this paper, we propose a simple paradigm DGDB for detecting graph database engine bugs. This paradigm offers a unique perspective in query generation for bug detection by integrating the cutting-edge Large Language Models with graph query languages. It firstly leverages powerful capabilities of ChatGPT in understanding and generating natural language to generate queries that meet specified constraints. Subsequently, it uses differential testing method to detect graph database engine bugs. We applied the paradigm

to cypher-based graph database engines and gremlin-based graph database engines, proposing DGDB-cypher and DGDB-gremlin to detect bugs in graph database engines. Experimental results demonstrate that our method is effective in discovering bugs in the latest versions of graph database engines for both query languages.

## REFERENCES

- [1] Ibrahim Abdelaziz, Essam Mansour, Mourad Ouzzani, Ashraf Aboulnaga, and Panos Kalnis. 2017. Query optimizations over decentralized RDF graphs. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 139–142.
- [2] DataStax Dylan Bethune-Waddell Expero Google Orion Health IBM Rafael Fernandes Robert Dale Seeq Amazon, Aurelius. 2023. JanusGraph. <https://janusgraph.org/>.
- [3] Sjoerd Mullender Andreas Seltenreich, Bo Tang. 2022. SQLsmith. <https://github.com/anse1/sqlsmith>.
- [4] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Computing Surveys (CSUR)* 40, 1 (2008), 1–39.
- [5] Dana Angluin. 1990. Negative results for equivalence queries. *Machine Learning* 5 (1990), 121–150.
- [6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [7] Donald D Chamberlin and Raymond F Boyce. 1974. SEQUEL: A structured English query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. 249–264.
- [8] Zihan Chen, Lei Nico Zheng, Cheng Lu, Jialu Yuan, and Di Zhu. 2023. ChatGPT Informed Graph Neural Network for Stock Movement Prediction. *arXiv preprint arXiv:2306.03763* (2023).
- [9] Edgar Frank Codd. 1983. A relational model of data for large shared data banks. *Commun. ACM* 26, 1 (1983), 64–69.
- [10] Jiayi Cui, Zongjian Li, Yang Yan, Bohua Chen, and Li Yuan. 2023. Chatlaw: Open-source legal large language model with integrated external knowledge bases. *arXiv preprint arXiv:2306.16092* (2023).
- [11] Andrzej Czerepinski. 2016. Application of graph databases for transport purposes. *Bulletin of the Polish Academy of Sciences. Technical Sciences* 64, 3 (2016), 457–466.
- [12] SAP Enterprise. 2023. OrientDB Community Edition. <http://www.orientdb.org/>.
- [13] Robert B Evans and Alberto Savoia. 2007. Differential testing: a new approach to change detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*. 549–552.
- [14] Apache Software Foundation. 2023. HugeGraph. <https://hugegraph.apache.org/>.
- [15] Apache Software Foundation. 2023. TinkerGraph. <https://tinkerpop.incubator.apache.org/>.
- [16] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Linddaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*. 1433–1445.
- [17] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. 2015. Fixing recurring crash bugs via analyzing q&a sites (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 307–318.
- [18] Marielet Guillermo, Maverick Rivera, Ronnie Concepcion, Robert Kerwin Biliones, Argel Bandala, Edwin Sybingco, Alexis Fillone, and Elmer Dadios. 2022. Graph Database-modelled Public Transportation Data for Geographic Insight Web Application. In *2022 IEEE/ACIS 23rd International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. IEEE, 2–7.
- [19] Edwin Hewitt and Leonard J Savage. 1955. Symmetric measures on Cartesian products. *Trans. Amer. Math. Soc.* 80, 2 (1955), 470–501.
- [20] Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, Wenpin Jiao, and Tao Xie. 2023. GDsmith: Detecting bugs in Cypher graph database engines. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.
- [21] Bitnine Global Inc. 2023. Agensgraph. <https://bitnine.net/agensgraph/>.
- [22] Neo4j Inc. 2023. Neo4j. <https://neo4j.com/>.
- [23] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund Clarke. 2005. Word level predicate abstraction and refinement for verifying RTL Verilog. In *Proceedings of the 42nd annual Design Automation Conference*. 445–450.
- [24] Matteo Kamm. 2022. *Testing Graph Databases using Predicate Partitioning*. Master’s thesis. ETH Zurich.
- [25] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C Berg, Wan-Yen Lo, et al. 2023. Segment anything. *arXiv preprint arXiv:2304.02643* (2023).
- [26] Takahiro Konno, Runhe Huang, Tao Ban, and Chuanhe Huang. 2017. Goods recommendation based on retail knowledge in a Neo4j graph database combined with an inference mechanism implemented in jess. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE, 1–8.
- [27] Bingfeng Luo, Yansong Feng, Zheng Wang, Songfang Huang, Rui Yan, and Dongyan Zhao. 2018. Marrying up regular expressions with neural networks: A case study for spoken language understanding. *arXiv preprint arXiv:1805.05588* (2018).
- [28] Hanjia Lyu, Song Jiang, Hanqing Zeng, Yinglong Xia, and Jiebo Luo. 2023. Llm-rec: Personalized recommendation via prompting large language models. *arXiv preprint arXiv:2307.15780* (2023).
- [29] Jun Ma and Bo Wang. 2023. Segment anything in medical images. *arXiv preprint arXiv:2304.12306* (2023).
- [30] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.
- [31] OpenAI. 2022. ChatGPT. <https://openai.com/blog/chatgpt/>.
- [32] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.
- [33] Rob Reagan and Rob Reagan. 2018. Cosmos DB. *Web Applications on Azure: Developing for Global Scale* (2018), 187–255.
- [34] Yuxiang Ren, Hao Zhu, Jiawei Zhang, Peng Dai, and Liefeng Bo. 2021. Ensemfdet: An ensemble approach to fraud detection based on bipartite graph. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2039–2044.
- [35] Manuel Rigger. 2022. SQLancer. <https://github.com/sqlancer/sqlancer>.
- [36] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1140–1152.
- [37] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.
- [38] Manuel Rigger and Zhendong Su. 2020. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 667–682.
- [39] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data*. "O’Reilly Media, Inc."
- [40] Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. 1–10.
- [41] Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. 1–10.
- [42] Sudipta Sen, Akash Mehta, Runa Ganguli, and Soumya Sen. 2021. Recommendation of influenced products using association rule mining: Neo4j as a case study. *SN Computer Science* 2 (2021), 1–17.
- [43] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. Citeseer, 618–622.
- [44] solid IT. 2023. DB-Engines Ranking of Graph DBMS. <https://db-engines.com/en/ranking/graph+dbms>.
- [45] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.
- [46] Ran Wang, Zhengyi Yang, Wenjie Zhang, and Xuemin Lin. 2020. An empirical study on recent graph database systems. In *Knowledge Science, Engineering and Management: 13th International Conference, KSEM 2020, Hangzhou, China, August 28–30, 2020, Proceedings, Part I* 13. Springer, 328–340.
- [47] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.
- [48] Jiawei Zhang. 2023. Graph-ToolFormer: To Empower LLMs with Graph Reasoning Ability via Prompt Augmented by ChatGPT. *arXiv preprint arXiv:2304.11116* (2023).
- [49] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).
- [50] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. 2022. Finding bugs in Gremlin-based graph database systems via randomized differential testing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 302–313.