2024 IEEE Conference on Software Testing, Verification and Validation (ICST)

Differential Optimization Testing of Gremlin-Based Graph Database Systems

Yingying Zheng[1,2], Wensheng Dou[1,2,3,4*], Lei Tang[1,2], Ziyu Cui[1,2], Jiansen Song[1,2], Ziyue Cheng[5],
Wei Wang[1,2,3,4], Jun Wei[1,2,3,4], Hua Zhong[1,2], Tao Huang[1,2]
[1]*State Key Lab of Computer Science at ISCAS*, [2]*University of CAS, Beijing, China*
[3]*Nanjing Institute of Software Technology*, [4]*University of CAS, Nanjing, China*
[5]*Huazhong University of Science and Technology, Hubei, China*
[1]{zhengyingying14, wsdou, tanglei20, cuiziyu20, songjiansen20, wangwei, wj, zhonghua, tao}@otcaix.iscas.ac.cn
[5]u202015309@hust.edu.cn

***Abstract*—Graph database systems (GDBs) allow efficiently creating, modifying, and retrieving graph data in a graph database. To accelerate graph queries, GDBs usually adopt various and complex optimization strategies. However, incorrect optimizations in GDBs can introduce *optimization bugs*, which cause a graph query to compute an incorrect query result, e.g., omitting a vertex in a graph database.**

In this paper, we propose *Differential Optimization Testing* (DOT), an effective and automated approach to detect optimization bugs in GDBs that adopt Gremlin as their query language. The main idea of DOT is that, given a Gremlin query Q, we execute it on the target GDB with two different optimization configurations and then verify whether they can compute the same query results for query Q. Any inconsistency between their query results indicates an optimization bug in the target GDB. To improve the efficiency of differential testing in DOT, we further propose an optimization-guided approach, aiming to explore more optimization strategies and more graph database features. We evaluate DOT on six popular and widely-used GDBs, i.e., Neo4j, OrientDB, JanusGraph, HugeGraph, TinkerGraph, and ArcadeDB. In total, we have found 28 unique optimization bugs, 16 of which have been confirmed as previously-unknown bugs.

***Index Terms*—graph database system, Gremlin, optimization bug, bug detection**

I. Introduction

Graph database systems (GDBs) (e.g., Neo4j [1], OrientDB [2], JanusGraph [3], TigerGraph [4], and NebulaGraph [5]) can efficiently store and retrieve graph data, and have played a significant role in many applications [6]–[9] (e.g., social networks [6]). Gremlin [10], as a popular graph query language, has been supported by about half of the GDBs (e.g., Neo4j, OrientDB, JanusGraph, and HugeGraph [11]) in DB-Engines Ranking for GDBs [12]. We refer to these GDBs that adopt Gremlin as their query language as *Gremlin-based GDBs*.

Gremlin-based GDBs support various optimization strategies to accelerate graph queries. For example, `FilterRankingStrategy` can reorder filtering operations in Gremlin queries to query graph data more efficiently, and `CountStrategy` can optimize counting operations by limiting the number of incoming elements. GDB users can configure their optimization strategies in their Gremlin queries. For example, in Fig. 1, we can execute the query of Line 3 with `LazyBarrierStrategy` off (Line 1).

```
1 g.withoutStrategies(LazyBarrierStrategy)    //off
2  .withoutStrategies(HugeVertexStepStrategy)//off
3  .E().bothV().not(__.in(’acting’))
4   -- v:{1,2,3,4} ✘
5
6 g.withStrategies(LazyBarrierStrategy)    //on
7  .withStrategies(HugeVertexStepStrategy)//on
8  .E().bothV().not(__.in(’acting’))
9   -- v:{1,3,4} ✔
```

Fig. 1. An optimization bug HugeGraph#2163 detected by our approach in HugeGraph [11]. When turning off the optimization strategies `LazyBarrierStrategy` and `HugeVertexStepStrategy` in HugeGraph, this query wrongly retrieves all vertices in Fig. 2.

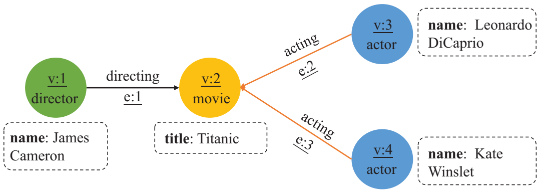


Fig. 2. A labeled property graph. We label each vertex and edge in the graph with unique IDs, e.g., v:1 represents a vertex with an ID 1.

The complexity of these optimization strategies can pose a major correctness challenge for Gremlin-based GDBs. Especially, incorrect optimizations in Gremlin-based GDBs can introduce *optimization bugs*, which cause GDBs to return incorrect query results (e.g., omitting a vertex in a graph database) for a given Gremlin query. Such silent optimization bugs are likely to go unnoticed by GDB developers.

Fig. 1 shows a real-world optimization bug HugeGraph#2163 that we detected in HugeGraph [11]. In this test case, we execute a Gremlin query of Line 3 (Line 8) with two different optimization configurations to retrieve the graph data shown in Fig. 2. Specifically, this Gremlin query first gets all edges (i.e., $E()$), then retrieves both incoming and outgoing vertices of each edge (i.e., $bothV()$), and finally removes the incoming vertices of the edges with

[*] CAS is the abbreviation of Chinese Academy of Sciences. ISCAS is the abbreviation of Institute of Software Chinese Academy of Sciences.

979-8-3503-0818-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ICST60714.2024.00012

label $acting$ (i.e., $not(\_\_.in('acting'))$). We execute this query in HugeGraph with two different optimization configurations, one with both the strategies `LazyBarrierStrategy` and `HugeVertexStepStrategy` off (Line 1-2), and the other with both the strategies `LazyBarrierStrategy` and `HugeVertexStepStrategy` on (Line 6-7). The correct result of this query should be v:{1, 3, 4} (Line 9). However, HugeGraph returns an incorrect result v:{1, 2, 3, 4} (Line 4) with the first optimization configuration (Line 1-2). HugeGraph developers explained that they forgot to remove the dirty edges of the vertices in the cache. HugeGraph developers have confirmed it as a previously-unknown bug and fixed it.

Recently, several approaches [13]–[16] have been proposed to find logic bugs in GDBs. Most of these approaches (e.g., Grand [13], GDsmith [16], and RD^2 [15]) utilize differential testing to find logic bugs in multiple GDBs. However, they can miss logic bugs when all target GDBs return incorrect query results. Many testing approaches [17]–[26] (e.g., TLP [20], NoREC [21], and DQE [25]) have been proposed to test relational database systems. Among these approaches, NoREC [21] can detect optimization bugs in relational database systems by rewriting an optimized SQL query into a non-optimizing SQL query. However, a Gremlin query cannot be rewritten into a non-optimizing Gremlin query by using the idea in NoREC because the *procedural* Gremlin queries in Gremlin-based GDBs adopt different query patterns from that of the *declarative* SQL queries in relational database systems. Thus, we still lack an effective approach to find optimization bugs in GDBs.

In this paper, we propose *D̲ifferential O̲ptimization T̲esting* (DOT), an automated approach to detect optimization bugs in Gremlin-based GDBs. We first randomly generate a graph database gdb. Based on the generated graph database gdb, we further randomly generate valid Gremlin queries. Then, for a generated Gremlin query Q, we generate candidate optimization configurations (an optimization configuration is a sequence of on and off states for optimization strategies supported by the target GDB), and execute Q on the graph database gdb with two different optimization configurations. Any inconsistency of their query results reveals an optimization bug in the target GDB.

To effectively detect and diagnose optimization bugs via DOT, we need to solve the following two technical challenges. First, randomly generated graph databases and Gremlin queries can potentially trigger the same optimization strategies, thus wasting testing efforts. To address this challenge, we propose an optimization-guided approach to filter out Gremlin queries that trigger the same optimization strategies, thus exploring more unique optimization strategies and further detecting more optimization bugs. Second, an optimization configuration usually contains optimization strategies that are not related to an optimization bug. It is time-consuming and error-prone to manually locate optimization strategies that cause an optimization bug. To solve this challenge, we provide an automated approach to locate faulty optimization strategies.

To the best of our knowledge, DOT is the first approach to detect optimization bugs in GDBs. We evaluate DOT on six widely-used Gremlin-based GDBs, i.e., Neo4j [1], OrientDB [2], JanusGraph [3], HugeGraph [11], TinkerGraph [27], and ArcadeDB [28]. At the time of writing this paper, we have found 28 unique optimization bugs in these six target GDBs. Among these 28 optimization bugs, 16 bugs have been verified as previously-unknown bugs, 5 out of which have been fixed by GDB developers. Our experimental results also show that our optimization-guided approach can detect 1.9x more bugs and 1.1x more unique bugs using 2.2x fewer test cases than the random approach, and our automated locating approach can accurately locate faulty optimization strategies. Furthermore, we compare DOT with the existing approaches, i.e., differential testing (i.e., Grand [13]) and query partitioning (i.e., GDBMeter [14]). For the 28 optimization bugs that DOT detected, 19 bugs are out of the scope of these approaches' detection capability, and cannot be detected by them. We have made DOT available at https://github.com/tcse-iscas/DOT.

To sum up, this paper makes the following contributions.

- We propose DOT for finding optimization bugs in Gremlin-based GDBs by identifying inconsistencies among different optimization configurations.
- To improve the efficiency of DOT, we propose an optimization-guided approach to explore more optimization strategies and graph database features.
- We evaluate DOT on six widely-used Gremlin-based GDBs. In total, we have detected 28 unique optimization bugs in them, 16 of which have been confirmed as new bugs.

II. Preliminaries

A. Graph Model and Gremlin Query Language

1) Graph Model: Most GDBs (e.g., Neo4j [1] and OrientDB [2]) are built on the *labeled property graph model* [29] to store and retrieve graph data. A labeled property graph model consists of vertices, their associated edges, and properties. Vertices and edges can be divided into different vertex and edge groups by their labels. Properties are used to describe attributes of different groups of vertices or edges. For example, in Fig. 2, this labeled property graph consists of four vertices (i.e., v:1, v:2, v:3, and v:4) and three edges (i.e., e:1, e:2, and e:3). These four vertices are divided into three groups, i.e., v:1 with label $director$, v:2 with label $movie$, and v:3 and v:4 with label $actor$, and have different properties, e.g., v:1 has a property $name$, while v:2 has a property $title$. These three edges are divided into two groups, i.e., e:1 with label $directing$, and e:2 and e:3 with label $acting$.

2) Gremlin Query Language: GDBs can create, modify, and retrieve graph data in labeled property graphs utilizing graph query languages, e.g., Gremlin [10], [30] developed by Apache TinkerPop [31], Cypher [32] developed by Neo4j [33], GSQL [34] in TigerGraph [35], and nGQL [36] in NebulaGraph [5], [37]. According to the DB-Engines Ranking [12], Gremlin, which is supported by about half of GDBs, has been one of the most popular graph query languages.

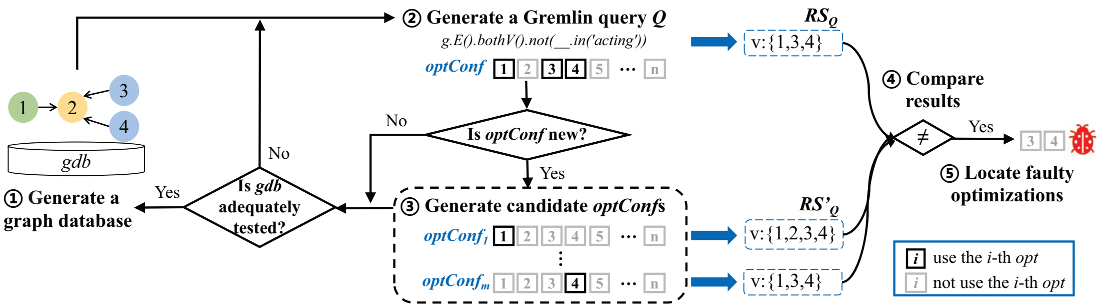


Fig. 3. Approach overview.

Different from *declarative* query languages (e.g., SQL in relational database systems), Gremlin is a *procedural* query language and allows developers to traverse labeled property graphs by linking a sequence of graph traversals (i.e., Gremlin APIs) after a Gremlin traversal source g. These Gremlin API calls can return vertices, edges, or properties, and some of them can act as nested sub-queries within other Gremlin APIs. For example, the Gremlin query in Fig. 1 consists of four Gremlin query APIs, i.e., $E()$, $bothV()$, $not()$, and $in()$. Here, $in()$ retrieves incoming vertices according to the given edge labels (e.g., $acting$), and acts as a nested sub-query in $not()$. Gremlin also supports a series of update APIs, e.g., adding vertices (edges) by $addV()$ ($addE()$) in a graph database, and dropping vertices (edges) by $V().drop()$ ($E().drop()$). We mainly test Gremlin query APIs in this paper.

B. Optimizations in Gremlin-Based GDBs

Gremlin-based GDBs utilize *Gremlin Traversal Machine* (GTM) [30] to process Gremlin queries. To efficiently execute a Gremlin query, GTM adopts various optimization strategies provided by Apache TinkerPop [31], and GDB-specific optimization strategies provided by each Gremlin-based GDB to determine the most optimal traversal execution steps according to the costs of querying graph data [38]. These optimization strategies applied to Gremlin-based GDBs should not change the execution semantics of Gremlin queries.

We obtain these optimization strategies provided by Apache TinkerPop and Gremlin-based GDBs from their source codes in GitHub. As far as we know, Apache TinkerPop supports 15 optimization strategies, which can be enabled in all Gremlin-based GDBs. Besides, Gremlin-based GDBs support their own strategies to adapt to GDB-specific optimizations. Specifically, JanusGraph provides 11 strategies. OrientDB and HugeGraph provide 3 strategies, respectively. TinkerGraph and ArcadeDB provide 2 strategies, respectively. Neo4j only provides one strategy. Note that we generate an optimization configuration including both the common optimization strategies and the GDB-specific optimization strategies for a GDB. For example, in TinkerGraph, we consider 17 optimization strategies in total.

GDB users can configure optimization strategies for their Gremlin queries. Specifically, we can turn on or off an optimization strategy by adding a $withStrategies()$ or a $wihtoutStrategies()$ configuration after the Gremlin traversal source g. For example, for the Gremlin query (Line 6-8) in Fig. 1, we turn on both strategies `LazyBarrierStrategy` and `HugeVertexStepStrategy` by adding a configuration $withStrategies(LazyBarrierStrategy)$ and a configuration $withStrategies(HugeVertexStepStrategy)$ after g. After that, GTM can execute Gremlin queries with these configured optimization strategies. Note that relational database systems adopt different optimization mechanisms [39]–[42], which cannot be easily configured by users.

III. Approach

In this paper, we propose *Differential Optimization Testing* (DOT), an effective and automated approach to reveal optimization bugs in Gremlin-based GDBs. Fig. 3 shows an overview of DOT. Specifically, we first randomly generate a graph database gdb (e.g., the graph database shown in Fig. 2) at ①. Then, we randomly generate a Gremlin query Q (e.g., the Gremlin query of Line 3 in Fig. 1) at ② and execute Q with an optimization configuration $optConf$ on the target GDB to retrieve its query result RS_Q (e.g., v:{1,3,4}).

After that, we generate candidate optimization configurations (e.g., $optConf_1$, ..., $optConf_m$) by enumerating the used optimization strategies of Gremlin query Q via combinatorial testing at ③. Given a generated optimization configuration (e.g., $optConf_1$), we execute Gremlin query Q with it on graph database gdb, and retrieve Q's query result RS'_Q (e.g., v:{1,2,3,4}). At ④, we compare the query result RS_Q with RS'_Q, and verify whether they are the same. If these two query results are inconsistent, an optimization bug is revealed. We then automatically locate the faulty optimization strategies that trigger this optimization bug at ⑤. Note that in an optimization configuration, the black (gray) box with a number i means that the i-th optimization strategy (*opt* for short in Fig. 3) of the target GDB is used (not used) when executing query Q.

We design an optimization approach to guide DOT to efficiently expose optimization bugs. Specially, if a generated Gremlin query Q triggers the same optimization strategies as some previous Gremlin queries, we re-execute ② to generate a new Gremlin query, aiming to explore new optimization strategies. Furthermore, if no unique optimization strategy is triggered on a generated graph database gdb for a period of time, we re-execute ① to generate a new graph database, assuming that this new generated graph database will subsequently lead to trigger new graph database features and unique optimization strategies, thus detecting new optimization bugs.

A. Generating Graph Databases and Gremlin Queries

Our graph database and Gremlin query generation are developed based on *Grand* [13]. Here, we only briefly explain how we generate them for completeness.

1) Generating Graph Databases: To generate a graph database, we first randomly generate its graph schema, i.e., vertex and edge types. Each vertex type consists of a vertex label name and a set of properties. Each edge type consists of an edge label name, an incoming vertex type, an outgoing vertex type and a set of properties. Each property type contains a property name and a data type.

We then randomly generate a number of detailed vertices and edges based on the generated graph schema. To generate a vertex, we randomly select a vertex type, and randomly generate its property values with the given data types, and then insert it into a graph database by executing Gremlin's update statement $g.addV(label).property(name, value)$. To generate an edge, we randomly select an edge type, two generated vertices according to its incoming and outgoing vertex types, and randomly generate its property values with the given data types. Then we insert the generated edge instance into a graph database by executing $g.addE(label).from(v1).to(v2).property(name, value)$.

We further randomly create some graph indexes for the properties of vertices and edges in the generated graph database. Specifically, for a randomly chosen property $prop$, we create graph indexes for property $prop$ using the specific syntaxes and index mechanisms of the target GDB.

2) Generating Gremlin Queries: We generate Gremlin queries guided by the Gremlin traversal model proposed in [13]. This traversal model defines graph traversal (i.e., Gremlin API) types and GDB users can link graph traversals correctly according to their output types. Given a maximum length $maxL$ of graph traversals, we first randomly select a traversal type and then randomly select the detailed graph traversal based on the traversal model, until the length $maxL$ is reached or a graph traversal that returns property values is selected. To generate query parameters, we select label and property names from the generated graph database, and select property values from the generated graph database or a random function.

Note that the generated database size and the maximum query length $maxL$ are configurable. In our experiment, we generate a graph database with 100 vertices and 200 edges, and set $maxL$ to 10, which we determine to work well empirically.

B. Generating Optimization Configurations

A target GDB includes a sequence of optimization strategies $opt =< o_1, o_2, ..., o_n >$, in which o_i represents the i-th optimization strategy and n is the number of the optimization strategies supported by the target GDB. Note that each optimization strategy has two states, i.e., on and off. Thus, for each optimization strategy in opt, we can turn it on or off to generate an optimization configuration $optConf =< s_1, s_2, ..., s_n >$, in which s_i represents the state of o_i. We set s_i to 1 (0) to turn o_i on (off).

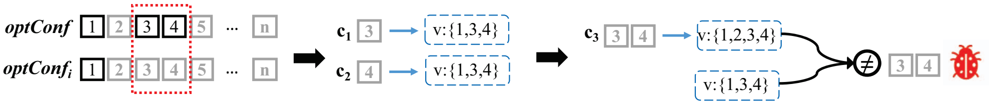
We suffer from the following two issues when generating optimization configurations. First, testing a Gremlin query with all combinations of optimization strategies would lead to a huge testing space. Although the number of optimization strategies in a target GDB is typically not large, e.g., 17 in TinkerGraph, we still need to generate 2^{17} optimization configurations to test all optimization strategies in TinkerGraph. Second, a completely random generation of optimization configurations can cause many useless configurations because different optimization strategies might be correlated with certain characteristics of the given Gremlin queries. These two issues can significantly affect the effectiveness of DOT.

To address the above two issues, we first analyze the correlations between a given query Q and certain optimization strategies in the target GDB. Our intuition is that an optimization strategy is worthy testing only if it is turned on and actually used in the execution of a query Q (used optimization strategies for short). Then, we utilize combinatorial testing [43] to enumerate the used optimization strategies of query Q and further generate candidate optimization configurations. Note that in our experiment, most optimization bugs can be triggered by only one optimization strategy. Therefore, we can adequately test optimization strategies using 2-wise combinatorial testing, i.e., for each pair of optimization strategies triggered by Q, we test all possible combinations of them.

Specifically, we first obtain the used optimization strategies by retrieving query Q's traversal explanation (see more in Section IV). Then, we generate pairwise combinations of the used optimization strategy by using the following algorithm [44]. (1) We generate all combinations for the used optimization strategies of query Q through Cartesian product, and store them in a candidate set $candidateSet$. (2) For each combination in $candidateSet$, we check whether all pairs of optimization strategies appear in other combinations of $candidateSet$. If yes, we discard it and continue the process. Following the above steps, we can obtain the pairwise combinations for query Q's used optimization strategies in $candidateSet$.

For example, to generate candidate optimization configurations for a Gremlin query Q in Fig. 3, we first obtain its used optimization strategies, e.g., o_1, o_3, and o_4. We then compute pairwise combinations of these optimization strategies, i.e., $< 1,1,1 >$, $< 1,0,0 >$, $< 0,1,0 >$, and $< 0,0,1 >$. Here, 1 (0) represents turning on (off) a strategy. Thus, the combination $< 1,0,0 >$ represents turning on a strategy o_1.

After that, we can generate candidate optimization configurations with combinations in $candidateSet$. Specifically, given a combination $optCom$ in $candidateSet$, for each used optimization strategy of query Q, we set its state to its corresponding value in $optCom$. For those unused optimization strategies of query Q, we set their states to 0. Thus, we can generate an optimization configuration with the combination



(a) Obtain *opts* with different states to *diff*

(b) Change *optConf* with one *opt* in *diff*

(c) Change *optConf* with two *opts* in *diff*

(d) Locate the faulty *opts*: *o$_3$* and *o$_4$*

Fig. 4. Locate faulty optimization strategies in Fig. 3.

$optCom$, e.g., $optConf_1 =< 1, 0, 0, 0, 0, ..., 0 >$ in Fig. 3. For a strategy o_i in a generated optimization configuration, we append a $withStrategies(o_i)$ ($withoutStrategies(o_i)$) after the Gremlin traversal source g when its state value s_i is 1 (0).

To perform differential testing for a given Gremlin query Q, we need to choose two different optimization configurations each time. In our implementation, we choose the optimization configuration that turning on all used optimization strategies as the original configuration $optConf$ in Fig. 3, and choose the other optimization configuration $optConf_i$ (e.g., $optConf_1$) from the generated candidate optimization configurations. We then execute query Q with $optConf$ and $optConf_i$ to retrieve its query result RS_Q and RS'_Q, respectively, and compare RS'_Q with RS_Q in Section III-C.

C. Comparing Query Results

For a Gremlin query Q, we compare its query result RS_Q under the optimization configuration $optConf$ with the query result RS'_Q under a generated optimization configuration $optConf_i$. Specifically, if both RS_Q and RS'_Q contain exception messages, we compare their reduced error messages (e.g., a substring of exception stack trace). If they belong to different error types, we consider that an optimization bug is detected. If one of RS_Q or RS'_Q contains an exception message, we also report it as an optimization bug. If none of RS_Q and RS'_Q contain exception messages, we first sort elements in them and then compare their elements one by one. Any inconsistency reveals an optimization bug. For example, in Fig. 3, we can report an optimization bug because the elements in RS_Q (i.e., v:{1,3,4}) are different from those in RS'_Q (i.e., v:{1,2,3,4}).

D. Locating Faulty Optimization Strategies

For each reported optimization bug, we locate the optimization strategies that trigger it. Algorithm 1 illustrates how we locate faulty optimization strategies for a given Gremlin query Q. Given the original optimization configuration $optConf$ and a new generated optimization configuration $optConf_i$ that triggers an optimization bug, we first obtain a sequence of optimization strategies $diff$ with different states between $optConf$ and $optConf_i$ (Line 1), e.g., $diff =< o_3, o_4 >$ in Fig. 4. After that, we iteratively enumerate optimization strategies in $diff$, generate new configurations by changing the states of the enumerated optimization strategies in $optConf$, and verify whether we can retrieve the same query result RS'_Q as RS_Q (Line 2-9). If not, we locate the minimum change of $optConf$, and then stop this process (Line 5-7).

Algorithm 1: Locate Faulty Optimization Strategies

```
   Input: $Q$ (A Gremlin query);
          $optConf$ (The original optimization configuration);
          $optConf_i$ (A new generated optimization configuration that
          triggers an optimization bug);
          $RS_Q$ (The query result of executing $Q$ with $optConf$)
   Output: $faultOpts$ (Faulty optimization strategies)
1  $diff \leftarrow$ getDifferentOpts($optConf, optConf_i$)
2  for $t \leftarrow 1; t \leq diff.size; t++$ do
3      foreach $opts \in$ enumDiff($diff, t$) do
4          $RS'_Q \leftarrow$ applyDiff($Q, opts$)
5          if $RS_Q \neq RS'_Q$ then
6              return $opts$
7      end
8  end
```

Specifically, as shown in Fig. 4 (b-d), we first iteratively enumerate one optimization strategy in $diff$ (e.g., $< o_3 >$ and $< o_4 >$), and apply its state on $optConf$. If we retrieve a different query result from RS_Q in an iteration, we locate the faulty optimization strategy and stop the process. If not, we continue to enumerate two optimization strategies in $diff$ (e.g., $< o_3, o_4 >$) and generate a new configuration by applying these two states of optimization strategies on $optConf$. If we retrieve a different query result with the new configuration, we locate the faulty optimization strategies, e.g., $< o_3, o_4 >$ in Fig. 4, and stop the process. If not, we continue to enumerate three and more optimization strategies until we find the faulty optimization strategies.

E. Optimizing Differential Testing

We optimize our differential testing by focusing on exploring new optimization strategies and new graph database features, thus exposing more optimization bugs.

1) Exploring New Optimization Strategies: In a generated graph database gdb, if a Gremlin query can trigger the same optimization strategies as some previously tested Gremlin queries, they are more likely to trigger the same optimization bugs. Therefore, we need to filter it out and generate a new Gremlin query to explore more unique optimization strategies.

Specifically, for a Gremlin query Q, we first obtain its used optimization strategies $usedOpts$. Then, we check whether the combination of these strategies in $usedOpts$ exists in the optimization set $optSet$. If yes, we think a similar Gremlin query has been tested, then discard query Q and generate a new one. Otherwise, we insert the combination of strategies in $usedOpts$ into the optimization set $optSet$ and generate candidate configurations for Q and test them.

```
1  g.addV('vL').property('vp', 11) // v1
2  g.addV('vL').property('vp', 21) // v2
3  g.addV('vL').property('vp', 31) // v3
4  g.addE('eL').from(v1).to(v2) // e1
5  g.addE('eL').from(v2).to(v3) // e2
6  g.addE('eL').from(v1).to(v3) // e3
7
8  g.withStrategies(LazyBarrierStrategy) // on
9   .V().in().has('vp', gt(10)) // -- v:{1,1,3}
10  .range(0,2) -- v:{1,1}
11
12 g.withoutStrategies(LazyBarrierStrategy) // off
13  .V().in().has('vp', gt(10)) // -- v:{1,3,1}
14  .range(0,2) -- v:{1,3}
```

Fig. 5. A corner case TinkerGraph#2933.

2) Exploring New Graph Databases: For a generated graph database gdb, if no unique optimization strategy is triggered for a period of time, then we can hardly detect new optimization bugs in gdb as it has been adequately tested. Therefore, we need to generate a new graph database, aiming to trigger new graph database features (e.g., new data type).

Specifically, for a Gremlin query Q, we first retrieve its used optimization strategies $usedOpts$ and then insert each of them into a unique set $uniqueSet$, in which we store unique optimization strategies. Given a used optimization strategy of query Q, we check whether it exists in $uniqueSet$, and insert it if not. If no new optimization strategy is inserted into $uniqueSet$ for a fixed number of queries, we consider that this graph database has been tested stressfully and invoke ① to generate a new graph database. Otherwise, we continue to test the target GDB using the same graph database gdb.

Note that this fixed query number is configurable. A higher query number means that we test a GDB using more Gremlin queries on a single graph database, while a lower query number indicates that we test a GDB using more graph databases. In our experiment, we set it to $1,000$.

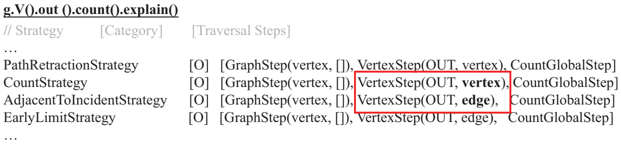
F. Corner Cases

Theoretically, our approach can test all optimization strategies supported by target GDBs because optimization strategies should not change the query results of Gremlin queries. However, there are two corner cases that we cannot test.

`LazyBarrierStrategy` in Apache TinkerPop is designed to optimize aggregated operations, e.g., $order()$ and $count()$. However, we find that this strategy sometimes can change the output order of graph data. As a result, Gremlin queries containing range operations, e.g., $range()$, may return different query results with `LazyBarrierStrategy` on and off. In Fig. 5, we create three vertices and three edges (Line 1-6), and query incoming vertices whose value of vp is greater than 10 (Line 9, 13). We obtain a query result v:$\{1, 1, 3\}$ with `LazyBarrierStrategy` on (Line 9), and a query result v:$\{1, 3, 1\}$ with `LazyBarrierStrategy` off (Line 13). Therefore, when we output the first two elements of these two query results using $range(0, 2)$, we retrieve two different results, i.e., v:$\{1, 1\}$ (Line 10) and v:$\{1, 3\}$ (Line 14).

```
1 g.addV().property('vp1', 11) // v1
2 g.addV().property('vp2', 'hello') // v2
3
4 g.withoutStrategies(ProductiveByStrategy) //off
5  .V().order().by('vp1')  -- v:{1}
6
7 g.withStrategies(ProductiveByStrategy) //on
8  .V().order().by('vp1')  -- v:{1,2}
```

Fig. 6. A corner case TinkerGraph#2900.



```
g.V().out ().count().explain()
// Strategy        [Category]      [Traversal Steps]
…
PathRetractionStrategy        [O]   [GraphStep(vertex, []), VertexStep(OUT, vertex), CountGlobalStep]
CountStrategy                 [O]   [GraphStep(vertex, []), VertexStep(OUT, vertex), CountGlobalStep]
AdjacentToIncidentStrategy    [O]   [GraphStep(vertex, []), VertexStep(OUT, edge),   CountGlobalStep]
EarlyLimitStrategy            [O]   [GraphStep(vertex, []), VertexStep(OUT, edge),   CountGlobalStep]
…
```

Fig. 7. Partial traversal explanation of a Gremlin query in TinkerGraph.

`ProductiveByStrategy` in Apache TinkerPop is designed to optimize $order().by(prop)$ that sorts vertices or edges by the given property name $prop$. This strategy forces a $Null$ value to be returned when the property $prop$ is absent. Therefore, it affects the query results when there are some vertices or edges without $prop$. For example, in Fig. 6, we sort vertices by property $vp1$ with `ProductiveByStrategy` off (Line 4) and on (Line 7). However, we obtain two different query results v:$\{1\}$ and v:$\{1{,}2\}$, because a $Null$ value is automatically generated for $vp1$ of v:2 with `ProductiveByStrategy` on.

IV. Implementations

We implement *DOT* on *Grand* [13]. In detail, we implement our method in around 900 lines of Java code, and adapt GDB-specific features in additional 300 lines of Java code, e.g., creating indexes.

Retrieving the used optimization strategies. For a given Gremlin query Q, we can obtain its used optimization strategies from Q's traversal explanation, which records the actual traversal steps performed by Q under each optimization strategy in a GDB. We fetch Q's traversal explanation by appending $explain()$ after it. As shown in Fig. 7, each line of the traversal explanation consists of three parts, i.e., strategy, category (e.g., optimization strategy $[O]$ and GDB-specific optimization strategy $[P]$), and traversal steps.

Note that the traversal steps of a used optimization strategy are different from that of its previous optimization strategy. Therefore, we can obtain the used optimization strategies by iteratively comparing the traversal steps of neighboring strategies in query Q's traversal explanation. In Fig. 7, `AdjacentToIncidentStrategy` has a traversal step `VertexStep(OUT,edge)`, which is different from the traversal step `VertexStep(OUT,vertex)` of its previous strategy `CountStrategy`. Thus, in this traversal explanation, a used optimization strategy of query Q is `AdjacentToIncidentStrategy`.

TABLE I
TARGET GDBS AND OPTIMIZATION BUGS DETECTED BY *DOT*

GDB	Ranking	GitHub	Detected Optimization Bugs					
		Stars	Detected	Fixed	Verified	Duplicate	False Positives	Unconfirmed
Neo4j	1	11.4k	2	0	0	2	0	0
OrientDB	5	4.6k	2	0	1	0	1	0
JanusGraph	7	4.8k	5	0	2	0	0	3
HugeGraph	24	2.3k	15	3	7	0	1	4
TinkerGraph	25	1.8k	3	2	0	1	0	0
ArcadeDB	28	309	1	0	1	0	0	0
Total	**-**	**-**	**28**	**5**	**11**	**3**	**2**	**7**

V. EVALUATION

To demonstrate the effectiveness of *DOT*, we evaluate *DOT* on six popular Gremlin-based GDBs, and try to answer the following three research questions:

- **RQ1:** How effective is *DOT* in detecting optimization bugs in real-world Gremlin-based GDBs?
- **RQ2:** Can *DOT* find optimization bugs more efficiently with our optimization-guided testing approach?
- **RQ3:** How does *DOT* perform compared with existing state-of-the-art approaches?

A. Experimental Setup

1) Target GDBs: We evaluate *DOT* on six widely-used Gremlin-based GDBs, i.e., Neo4j [1], OrientDB [2], JanusGraph [3], HugeGraph [11], TinkerGraph [27], and ArcadeDB [28]. The second and third columns in Table I show their DB-Engines Ranking for GDBs [12] and GitHub stars. We can see that these GDBs are all important and popular GDBs. Specifically, JanusGraph, HugeGraph, and TinkerGraph support graph model and can utilize Gremlin natively. OrientDB and ArcadeDB support multiple data models, including graph model, document, and key-value stores, etc, and implement their own TinkerPop [31] interfaces. We access graph databases in Neo4j through the Neo4j-Gremlin plugin [45], which is developed by Apache TinkerPop. In summary, the six target GDBs used in our experiment cover different kinds of GDBs and are representative.

We test the latest release versions of these target GDBs when we start this work, i.e., Neo4j-Gremlin 3.6.2 (with Neo4j 3.4.11), OrientDB 3.2.16, JanusGraph 0.6.3, HugeGraph 1.0.0, TinkerGraph 3.6.2, and ArcadeDB 23.2.1. Note that Neo4j 3.4.11 is not the latest version of Neo4j, but is the version used in the latest Neo4j-Gremlin plugin.

2) Experimental Infrastructure: We conduct all experiments on a machine with an Intel(R) Core(TM) i9-9900 processor, which has 16 physical and 32 logical cores clocked at 3.10GHz. Our test machine uses a 64-bit CentOS Linux release 7.7.1908 system with 8 GB RAM.

B. Detected Bugs

1) Testing Methodology: We run *DOT* on the six target GDBs. In each testing round, we run *DOT* to test a target GDB (e.g., TinkerGraph) within a time budget (e.g., five minutes in our experiment), and then manually analyze, reproduce, and filter out duplicate optimization bugs for the optimization bugs reported by *DOT*. After that, we continue to test this target GDB aiming to find more optimization bugs. We have tested each target GDB in 10 testing rounds in our experiment.

Specially, for a reported optimization bug, we manually reduce the Gremlin query and the graph database to a smaller one. Then, to distinguish unique bugs, we check whether the query patterns of the reduced Gremlin query are similar or exception messages are similar to the previously found bugs. If yes, we filter it out. Otherwise, we consider it as unique.

Note that for the strategies `LazyBarrierStrategy` and `ProductiveByStrategy` that affect the results of Gremlin queries, we do not test them in subsequent testing. However, for the strategies that trigger a true bug, we continue to test them because they may trigger different optimization bugs.

2) Bug Overview: In our experiment, *DOT* reports 18,481 bug reports in the six target GDBs. It is time-consuming to investigate all these bug reports. Therefore, we take a stratified random sample approach to sample a subset of representative bug reports for each faulty optimization strategy to identify unique optimization bugs. In each testing round, we first locate the faulty optimization strategies for each bug report by using the approach in Section III-D, and then randomly sample 50 bug reports for each located faulty optimization strategy combination in each GDB. If a located optimization strategy combination contains less than 50 bug reports, we keep all of them. We finally sample 2,549 bug reports for further investigation in 10 testing rounds. We investigate these bug reports and remove duplicate ones from these 2,549 bug reports. Finally, we obtain 28 unique optimization bugs in the six target GDBs, and submit them to their corresponding communities and wait for feedbacks from GDB developers.

Table I shows the overall bug statistics for the 28 unique optimization bugs. Out of the 28 optimization bugs, 16 bugs have been confirmed as new bugs, 5 of which have been fixed by GDB developers. For the remaining bugs, 3 bugs are considered duplicates to existing bug reports, 2 bugs are considered as false positives by GDB developers, and 7 bugs have not received any feedbacks from GDB developers yet.

False positives. Two bugs are considered as not bugs by GDB developers. In OrientDB#9885, OrientDB forgets to throw an exception when sorting vertices and edges for a non-existent property. OrientDB developers explained that OrientDB allows to access a non-existent property and just

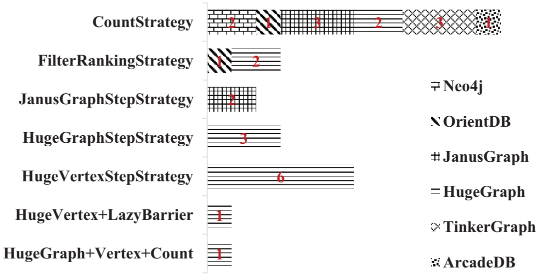


Fig. 8. The statistics of faulty optimization strategies.

returns a $Null$ value. However, OrientDB actually sometimes throws an exception, but sometimes returns a $Null$ value. Therefore, we still believe that OrientDB incorrectly handles $order()$ operation. In HugeGraph#1966, HugeGraph returns an incorrect result when querying vertices or edges by filtering properties using $not(eq())$. HugeGraph developers explained that HugeGraph now does not support $not(eq())$ with a graph index and they plan to support it in the future.

> ***RQ1:*** *DOT is effective in finding optimization bugs. We have detected 28 unique optimization bugs, 16 of which are previously-unknown bugs.*

3) Optimization Bug Analysis: For each optimization bug, we locate its faulty optimization strategies. We further analyze the 28 optimization bugs, and obtain the statistics about their faulty optimization strategies.

The involved faulty optimization strategies. As shown in Fig. 8, the majority of (26 out of 28) optimization bugs are triggered by only one optimization strategy. That is, most optimization strategies are independent. Specifically, among these 26 bugs, 15 bugs are triggered by the common optimization strategies provided by Apache TinkerPop, e.g., `CountStrategy` and `FilterRankingStrategy`. The remaining 11 bugs are triggered by the GDB-specific optimization strategies provided by target GDBs, 9 bugs of which occur in HugeGraph and the other 2 bugs in JanusGraph.

The remaining two bugs are triggered by more than one optimization strategies. One bug (Fig. 1) is triggered by turning off `HugeVertexStepStrategy` and `LazyBarrierStrategy` (`HugeVertex+LazyBarrier` for short). In this bug, `HugeVertexStepStrategy` and `LazyBarrierStrategy` affect each other, and cause HugeGraph to return an incorrect query result. Another bug is triggered by turning off three optimization strategies provided by HugeGraph (`HugeGraph+Vertex+Count` for short).

The states of faulty optimization strategies. As shown in Fig. 9, we count the states (on or off) of faulty optimization strategies. Surprisingly, more than half (16) of these 28 unique bugs are triggered by turning off optimization strategies. Among these 16 bugs, besides three bugs triggered by the common strategy `FilterRankingStrategy`, 13 bugs are caused by incorrect GDB-specific optimizations. For example, in Fig. 13, when we turn off the strategy `HugeVertexStepStrategy`, HugeGraph omits vertices in the graph database and returns an incorrect result. These bugs indicate that GDB developers have performed adequate testing when turning on these optimization strategies, but they neglect to test GDBs when these optimization strategies are turned off.

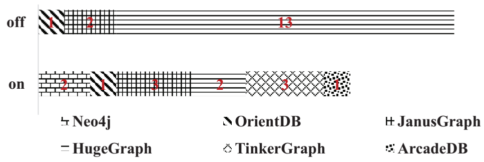


Fig. 9. The states of faulty optimization strategies.

TABLE II
COMPARISON BETWEEN DOT AND DOT_{rand}

GDB	All bugs		Unique bugs		Test queries	
	DOT	DOT_{rand}	DOT	DOT_{rand}	DOT	DOT_{rand}
Neo4j	272	119	2	2	1,248	4,000
OrientDB	398	180	2	2	1,608	4,317
JanusGraph	327	173	5	5	1,444	2,797
HugeGraph	215	238	11	8	598	1,000
TinkerGraph	439	134	3	3	2,012	4,208
ArcadeDB	191	125	1	1	1,927	3,140
Total	**1,842**	**969**	**24**	**21**	**8,837**	**19,462**

The remaining 12 optimization bugs are triggered by turning on certain optimization strategies. For example, in Fig. 12, `CountStrategy` computes a wrong result when the filter condition in a predicate is abnormal.

> *Most optimization bugs are caused by independent optimization strategies. Both turning on and off optimization strategies can effectively expose optimization bugs.*

C. Efficiency of Optimization-Guided Testing Approach

To evaluate the efficiency of our optimization-guided approach in DOT, we construct a variant of DOT, i.e., DOT_{rand}, which removes the optimization approach from DOT. We evaluate whether DOT can find optimization bugs faster than DOT_{rand}. To fairly compare them, for each generated graph database, we randomly generate a fixed number (e.g., 1,000 in our experiment) of queries in DOT_{rand}.

To this end, we run DOT and DOT_{rand} to test each target GDB within a time budget (e.g., five minutes in our experiment). For all reported optimization bugs, we distinguish unique bugs by checking whether the query patterns or exception messages are similar to the previously found bugs.

Table II shows the experimental comparison results between DOT and DOT_{rand}. Overall, DOT can find 1.9x more bugs and 1.1x more unique bugs with 2.2x fewer test queries than DOT_{rand}. It is not surprising that DOT and DOT_{rand} can find all optimization bugs in most target GDBs except HugeGraph because optimization bugs detected in these GDBs do not require complex query patterns or complex graph databases. However, in HugeGraph, some optimization bugs require special query patterns (e.g., comparing a String value using $gte()$) or special graph database features (e.g., graph

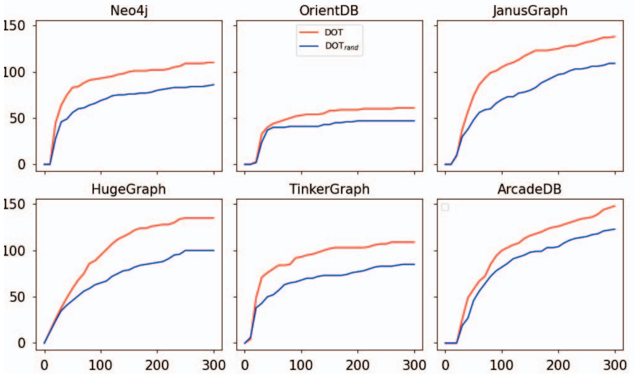


Fig. 10. The number of unique optimization combinations within 5 minutes (300 seconds in the x axis) running of DOT and DOT_{rand}.

data contains a Double value $Infinity$). DOT can easily and quickly expose these bugs because it can explore more unique optimization strategies and more graph database features.

Further, we count the number of unique combinations of optimization strategies within five minutes running of DOT and DOT_{rand}. As shown in Fig. 10, DOT can explore 1.3x more unique optimization combinations than DOT_{rand}, demonstrating the contribution of our optimization-guided approach. These results further illustrate that optimization bugs can be more efficiently detected by exploring more unique optimization strategies.

> ***RQ2:*** *DOT can explore more unique optimization strategies, thus exposing more optimization bugs than* DOT_{rand}.

D. Comparison with Existing Approaches

To the best of our knowledge, four existing approaches [13]–[16] can detect logic bugs in GDBs. Specifically, three (i.e., *Grand* [13], *GDsmith* [16], and *RD*^{2} [15]) use differential testing on multiple GDBs, and the remaining one (i.e., *GDBMeter* [14]) uses query partitioning. Therefore, we compare DOT with differential testing and query partitioning.

Comparison with differential testing. *Grand* applies differential testing to detect logic bugs in GDBs. It generates the same graph database and executes the same Gremlin queries in multiple target GDBs to verify whether they can return the same results. Any discrepancy reveals a logic bug. To compare DOT with differential testing (e.g., *Grand*), for each test case in the 28 optimization bug reports, we run it on our six target GDBs and verify whether their query results are the same. Any discrepancy will be considered as an optimization bug detected by *Grand*. As a result, among the 28 optimization bugs, *Grand* can detect 9 bugs. In summary, *Grand* cannot detect the remaining 19 optimization bugs that DOT detected.

Comparison with query partitioning. *GDBMeter* applies query partitioning [20] to test GDBs. Specifically, it partitions a query into three disjoint queries in which a predicate is evaluated to $true$, $false$ and $null$, respectively. Since we cannot successfully run *GDBMeter* in practice and *GDBMeter* only supports testing one Gremlin-based GDB (i.e., JanusGraph), we compare DOT with *GDBMeter* by applying query partitioning to each test case in the 28 bug reports. Specifically, we try to generate a $has(key)$ or revise a $has()$ operation in the original query, and then generate a predicate p to construct three disjoint sub-queries $has(key, p)$, $has(key, \neg p)$ and $hasNot(key)$. We then check whether the union of the query results computed by these three disjoint sub-queries is the same as the result of the original query. If not, then we consider that this bug can be detected by *GDBMeter*. As a result, among the 28 optimization bugs, only 2 bugs can be found by *GDBMeter*, which can also be detected by *Grand*.

```
1  v1 = g.addV('vL'); // v1
2  v2 = g.addV('vL'); // v2
3
4  g.withStrategies(CountStrategy) // on
5   .V().where(__.in().count().is(eq(-3)));
6   -- {Not a legal range: [0, -2]} ✘
7
8  g.withoutStrategies(CountStrategy) // off
9   .V().where(__.in().count().is(eq(-3)));
10  -- {} ✔
```

Fig. 11. A test case that triggers bug TinkerGraph#2891.

> ***RQ3:*** *DOT can detect new optimization bugs that existing approaches cannot find. Specifically, 19 out of 28 optimization bugs cannot be detected by existing approaches.*

E. Interesting Bugs

We have explained HugeGraph#2163 in Fig. 1. We further explain more newly detected optimization bugs in detail.

TinkerGraph#2891. Fig. 11 shows an optimization bug detected in TinkerGraph. The graph database consists of two vertices, i.e., v:1 and v:2 (Line 1-2). We retrieve vertices whose incoming vertex number is equal to a negative value -3 with `CountStrategy` on (Line 4-5). We expect TinkerGraph to return an empty set since the number of incoming vertex should be greater than or equal to zero. However, an unexpected exception is thrown (Line 6). We can get the correct result with `CountStrategy` off (Line 8-10). TinkerGraph developers explained that `CountStrategy` cannot correctly compare negative values with counted number and fixed it.

TinkerGraph#2893. Fig. 12 shows an optimization bug in TinkerGraph. The graph database consists of two vertices, i.e., v:1 and v:2, and one edge e:1 (Line 1-3). We first get all vertices, and then retrieve vertices whose outgoing vertex number is less than 1 or greater than 0. TinkerGraph returns an incorrect result v:{2} with `CountStrategy` on (Line 5-7). However, we can retrieve the correct result v:{1, 2} when we turn off the strategy `CountStrategy` (Line 9-11). TinkerGraph developers explained that the boundary values need to be better checked in `CountStrategy` and fixed it.

HugeGraph#2121. Fig. 13 shows an optimization bug detected in HugeGraph. In this test case, the graph database

```
v1 = g.addV('vL'); // v1
v2 = g.addV('vL'); // v2
e1 = g.addE('eL').from(v1).to(v2) // e1

g.withStrategies(CountStrategy) // on
 .V().where(__.out().count().is(outside(1,0)))
  -- v:{2} ✘

g.withoutStrategies(CountStrategy) // off
 .V().where(__.out().count().is(outside(1,0)))
  -- v:{1,2} ✔
```

Fig. 12. A test case that triggers bug TinkerGraph#2893.

```
 1  v1=g.addV('vL'); // v1
 2  v2=g.addV('vL'); // v2
 3  v3=g.addV('vL'); // v3
 4  e1=g.V(v2).addE('eL1').to(v1).property('p', 3);
 5  e2=g.V(v3).addE('eL2').to(v1).property('p', 2);
 6
 7  g.withoutStrategies(HugeVertexStepStrategy)//off
 8   .E().inV().inE('eL2');
 9   -- {} ✘
10
11  g.withStrategies(HugeVertexStepStrategy) // on
12   .E().inV().inE('eL2')
13   -- e:{2,2} ✔
```

Fig. 13. A test case that triggers bug HugeGraph#2121.

consists of 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 first retrieve incoming vertices of all edges (i.e., $E().inV()$) and then retrieve these vertices' incoming edges with an edge label $eL2$ (i.e., $inE('eL2')$) (Line 8). We obtain an incorrect result, i.e., an empty set, when we turn off the strategy `HugeVertexStepStrategy` (Line 7-9). However, we can obtain a correct result e:$\{2, 2\}$ with `HugeVertexStepStrategy` on (Line 11-13). HugeGraph developers have confirmed it and try to fix it.

VI. Discussion

Threats to validity. First, we evaluate DOT on six target GDBs. These GDBs are all well maintained and rank on the top of GDB popularity. Thus, we believe our target GDBs are representative. Second, manually filtering out duplicate optimization bugs may introduce human errors and omit real optimization bugs. To mitigate this threat, three authors carefully study the sampled bug reports and reach a consensus for them. Third, we may introduce threats when comparing DOT with existing approaches. To mitigate this threat, we run the available tool and make a careful manual analysis for them in order to compare them as fairly as possible.

Limitations. First, DOT can effectively detect optimization bugs caused by incorrect implementations of optimization strategies, but cannot test other logic bugs that are not related to optimization strategies. Second, DOT cannot test some optimization strategies (e.g., `ProductiveByStrategy`) that can affect the query results when turing them on or off. Third, DOT cannot automatically remove duplicate bugs because the manifestations of bugs for the same root cause are diverse and complex, which poses a challenge for automatically removing duplicate bugs. We will address this limitation in the future.

VII. Related Work

Testing graph database systems (GDBs). Recently, many works [13]–[16] have been proposed to test the correctness of GDBs. Grand [13], RD^2 [15], and GDsmith [16] detect logic bugs in multiple GDBs by utilizing differential testing [46]–[48], while GDBMeter [14] utilize metamorphic testing [49] to find logic bugs in an individual GDB. These works cannot effectively detect optimization bugs in GDBs.

Testing relational database systems (DBMSs). Researchers have developed various testing approaches on relational DBMSs [17]–[25], [50]–[56]. Some approaches [18], [19], [23], [50] (e.g., RAGS [18]) applied differential testing to test relational DBMSs. Some approaches [20], [21], [25] (e.g., NoREC [21]) utilizes metamorphic testing for finding logic bugs in relational DBMSs. Query generation [17], [54] (e.g., SQLsmith [17]) and generic fuzzing approaches (e.g., AFL [53]) can also be used to detect bugs in relational DBMSs. All the above testing approaches target relational DBMSs with *declarative* query language (i.e., SQL), which has totally different query patterns and syntaxes from the *procedural* Gremlin query language. Therefore, they cannot be applied to find optimization bugs in GDBs.

Compiler optimization testing. In compiler testing [57], many approaches [58]–[65] have been proposed to test compiler optimizers. Some works (e.g., [59], [64]) randomly generate compiler test cases to detect optimization bugs in compilers. Some works (e.g., [61], [63]) test compiler optimizers by generating equivalent programs. COTest [60] boosts compiler testing by exploring more optimizations. However, these approaches cannot be applied to test optimization bugs in GDBs, because they target totally different objects and goals.

VIII. Conclusion

Graph database systems suffer from optimization bugs caused by incorrect optimizations. In this paper, we propose DOT, an automated testing approach for finding optimization bugs via identifying inconsistencies between two different optimization configurations in a Gremlin-based graph database system. We further propose an optimization-guided approach to expose more optimization bugs quickly. We evaluate DOT on six representative Gremlin-based graph database systems, and have detected 28 unique optimization bugs, 16 of which have been verified as new bugs by GDB developers. We expect that the effectiveness of our approach can greatly improve the robustness of graph database systems.

Acknowledgment

This work was partially supported by National Natural Science Foundation of China (62072444), Major Project of ISCAS (ISCAS-ZD-202302), and Youth Innovation Promotion Association at Chinese Academy of Sciences (Y2022044). Wensheng Dou is the corresponding author.

REFERENCES

[1] "Neo4j," https://neo4j.com/, 2023.

[2] "Orientdb," https://orientdb.org, 2023.

[3] "Janusgraph," https://janusgraph.org, 2023.

[4] A. Deutsch, Y. Xu, M. Wu, and V. E. Lee, "Aggregation support for modern graph analytics in TigerGraph," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2020, pp. 377–392.

[5] "Open source, distributed, scalable, lightning fast," https://nebula-graph.io/, 2023.

[6] O. Erling, A. Averbuch, J. L. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat-Pérez, M. Pham, and P. A. Boncz, "The LDBC social network benchmark: Interactive workload," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015, pp. 619–630.

[7] Y. Ren, H. Zhu, J. Zhang, P. Dai, and L. Bo, "EnsemFDet: An ensemble approach to fraud detection based on bipartite graph," in *Proceedings of International Conference on Data Engineering (ICDE)*, 2021, pp. 2039–2044.

[8] B. Liu, X. Wang, P. Liu, S. Li, Q. Fu, and Y. Chai, "Unikg: A unified interoperable knowledge graph database system," in *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2021, pp. 2681–2684.

[9] M. Arenas, C. Gutiérrez, and J. F. Sequeda, "Querying in the age of graph databases and knowledge graphs," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2021, pp. 2821–2828.

[10] "Gremlin query language," https://tinkerpop.apache.org/gremlin.html, 2023.

[11] "Hugegraph," https://hugegraph.github.io/hugegraph-doc/, 2023.

[12] "Db-engines ranking of graph dbms," https://db-engines.com/en/ranking/graph+dbms, 2023.

[13] Y. Zheng, W. Dou, Y. Wang, Z. Qin, L. Tang, Y. Gao, D. Wang, W. Wang, and J. Wei, "Finding bugs in Gremlin-based graph database systems via randomized differential testing," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, 2022, pp. 302–313.

[14] M. Kamm, M. Rigger, C. Zhang, and Z. Su, "Testing graph database engines via query partitioning," in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023.

[15] R. Yang, Y. Zheng, L. Tang, W. Dou, W. Wang, and J. Wei, "Randomized differential testing of RDF stores," in *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE Demo)*, 2023, pp. 136–140.

[16] Z. Hua, W. Lin, L. Ren, Z. Li, L. Zhang, and T. Xie, "GDsmith: Detecting bugs in cypher graph database engines," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 163–174.

[17] "Sqlsmith," https://github.com/anse1/sqlsmith, 2023.

[18] D. S. Slutz, "Massive stochastic testing of SQL," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 1998, pp. 618–622.

[19] J. Jung, H. Hu, J. Arulraj, T. Kim, and W. Kang, "APOLLO: Automatic detection and diagnosis of performance regressions in database systems," *Proceedings of the VLDB Endowment (VLDB)*, vol. 13, no. 1, pp. 57–70, 2019.

[20] M. Rigger and Z. Su, "Finding bugs in database systems via query partitioning," in *Proceedings of ACM Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2020, pp. 211:1–30.

[21] ——, "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)*, 2020, pp. 1140–1152.

[22] ——, "Testing database engines via pivoted query synthesis," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 667–682.

[23] Z. Cui, W. Dou, Q. Dai, J. Song, W. Wang, J. Wei, and D. Ye, "Differentially testing database transactions for fun and profit," in *Proceedings of International Conference on Automated Software Engineering (ASE)*, 2022, pp. 35:1–35:12.

[24] J. Ba and M. Rigger, "Testing database engines via query plan guidance," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2023, pp. 2060–2071.

[25] J. Song, W. Dou, Z. Cui, Q. Dai, W. Wang, J. Wei, H. Zhong, and T. Huang, "Testing database systems via differential query execution," in *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023, pp. 2072–2084.

[26] W. Dou, Z. Cui, Q. Dai, J. Song, D. Wang, Y. Gao, W. Wang, J. Wei, L. Chen, H. Wang, H. Zhong, and T. Huang, "Detecting isolation bugs via transaction oracle construction," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2023, pp. 1123–1135.

[27] "Tinkergraph," https://github.com/tinkerpop/blueprints/wiki/tinkergraph, 2023.

[28] "The next generation multi-model database supporting graphs key/value, documents and time-series," https://arcadedb.com/, 2023.

[29] R. Wang, Z. Yang, W. Zhang, and X. Lin, "An empirical study on recent graph database systems," in *Proceedings of International Conference on Knowledge Science, Engineering and Management (KSEM)*, 2020, pp. 328–340.

[30] M. A. Rodriguez, "The gremlin graph traversal machine and language (invited talk)," in *Proceedings of the Symposium on Database Programming Languages*, 2015, pp. 1–10.

[31] "Tinkerpop," https://tinkerpop.apache.org/, 2023.

[32] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2018, pp. 1433–1445.

[33] Z. He, J. Yu, and B. Guo, "Execution time prediction for Cypher queries in the Neo4j database using a learning approach," *Symmetry*, vol. 14, no. 1, p. 55, 2022.

[34] A. Deutsch, "Querying graph databases with the GSQL query language," in *Proceedings of Simpósio Brasileiro de Banco de Dados (SBBD)*, 2018, p. 313.

[35] A. Deutsch, Y. Xu, M. Wu, and V. E. Lee, "Aggregation support for modern graph analytics in TigerGraph," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2020, pp. 377–392.

[36] "Nebula graph query language (nGQL)," https://docs.nebula-graph.io/2.0.1/3.ngql-guide/1.nGQL-overview/1.overview/, 2023.

[37] M. Wu, X. Yi, H. Yu, Y. Liu, and Y. Wang, "Nebula graph: An open source distributed graph database," *CoRR*, vol. abs/2206.07278, 2022.

[38] "Gremlin traversal strategy," https://tinkerpop.apache.org/docs/3.5.2/, 2023.

[39] B. Ding, S. Das, W. Wu, S. Chaudhuri, and V. R. Narasayya, "Plan stitch: Harnessing the best of many plans," *Proceedings of the VLDB Endowment (VLDB)*, vol. 11, no. 10, pp. 1123–1136, 2018.

[40] T. Neumann and B. Radke, "Adaptive optimization of very large join queries," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2018, pp. 677–692.

[41] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao, "Towards a learning optimizer for shared clouds," *Proceedings of the VLDB Endowment (VLDB)*, vol. 12, no. 3, pp. 210–222, 2018.

[42] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," *Proceedings of the VLDB Endowment (VLDB)*, vol. 12, no. 11, pp. 1705–1718, 2019.

[43] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys*, vol. 43, no. 2, pp. 11:1–11:29, 2011.

[44] K. Tai and Y. Lei, "A test generation strategy for pairwise testing," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 1, pp. 109–111, 2002.

[45] "Neo4j-gremlin," https://github.com/thinkaurelius/neo4j-gremlin-plugin, 2023.

[46] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[47] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of International Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 283–294.

[48] S. A. Chowdhury, S. Mohian, S. Mehra, S. Gawsane, T. T. Johnson, and C. Csallner, "Automatically finding bugs in a commercial cyber-physical system development tool chain with SLforge," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2018, pp. 981–992.

[49] M. N. Mansur, M. Christakis, and V. Wüstholz, “Metamorphic testing of Datalog engines,” in *Proceedings of Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, pp. 639–650.

[50] X. Liu, Q. Zhou, J. Arulraj, and A. Orso, “Automatic detection of performance bugs in database systems using equivalent queries,” in *Proceedings of International Conference on Software Engineering (ICSE)*, 2022, pp. 225–236.

[51] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid, “Query-aware test generation using a relational constraint solver,” in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008, pp. 238–247.

[52] S. A. Khalek and S. Khurshid, “Automated SQL query generation for systematic testing of database engines,” in *Proceedings of Automated Software Engineering (ASE)*, 2010, pp. 329–332.

[53] “Afl,” https://github.com/google/AFL, 2023.

[54] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, “SQUIRREL: Testing database management systems with language validity and coverage feedback,” in *Proceedings of Computer and Communications Security (CCS)*, 2020, pp. 58–71.

[55] Z. Hao, Q. Huang, C. Wang, J. Wang, Y. Zhang, R. Wu, and C. Zhang, “Pinolo: Detecting logical bugs in database management systems with approximate query synthesis,” in *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, 2023, pp. 345–358.

[56] Z. Cui, W. Dou, Y. Gao, D. Wang, J. Song, Y. Zheng, T. Wang, R. Yang, K. Xu, Y. Hu, J. Wei, and T. Huang, “Understanding transaction bugs in database systems,” in *Proceedings of International Conference on Software Engineering (ICSE)*, 2024.

[57] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, “A survey of compiler testing,” *ACM Computing Surveys*, vol. 53, no. 1, pp. 4:1–4:36, 2021.

[58] J. M. Caron and P. A. Darnell, “Bugfind: A tool for debugging optimizing compilers,” *ACM SIGPLAN Notices*, vol. 25, no. 1, pp. 17–22, 1990.

[59] C. J. Burgess and M. Saidi, “The automatic generation of test cases for optimizing fortran compilers,” *Information and Software Technology (IST)*, vol. 38, no. 2, pp. 111–119, 1996.

[60] J. Chen and C. Suo, “Boosting compiler testing via compiler optimization exploration,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 72:1–72:33, 2022.

[61] M. Sassa and D. Sudosa, “Experience in testing compiler optimizers using comparison checking,” in *Proceedings of Software Engineering Research and Practice & Conference on Programming Languages and Compilers (SERP)*, vol. 2, 2006, pp. 837–843.

[62] V. Le, C. Sun, and Z. Su, “Randomized stress-testing of link-time optimizers,” in *Proceedings of Symposium on Software Testing and Analysis (ISSTA)*, 2015, pp. 327–337.

[63] V. Le, M. Afshari, and Z. Su, “Compiler validation via equivalence modulo inputs,” in *Proceedings of Programming Language Design and Implementation (PLDI)*. ACM, 2014, pp. 216–226.

[64] A. Hashimoto and N. Ishiura, “Detecting arithmetic optimization opportunities for C compilers by randomly generated equivalent programs,” *IPSJ Transactions on System and LSI Design Methodology*, vol. 9, pp. 21–29, 2016.

[65] V. Le, C. Sun, and Z. Su, “Finding deep compiler bugs via guided stochastic program mutation,” in *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA)*, 2015, pp. 386–399.