GDsmith: Detecting Bugs in Cypher Graph Database Engines

Ziyue Hua
2201111637@stu.pku.edu.cn
Peking University
Beijing, China

Wei Lin
linwei@stu.pku.edu.cn
Peking University
Beijing, China

Luyao Ren
rly@pku.edu.cn
Peking University
Beijing, China

Zongyang Li
lizongyang@stu.pku.edu.cn
Peking University
Beijing, China

Lu Zhang
zhanglucs@pku.edu.cn
Peking University
Beijing, China

Wenpin Jiao
jwp@sei.pku.edu.cn
Peking University
Beijing, China

Tao Xie*
taoxie@pku.edu.cn
Peking University
Beijing, China

ABSTRACT

Graph database engines stand out in the era of big data for their efficiency of modeling and processing linked data. To assure high quality of graph database engines, it is highly critical to conduct automatic test generation for graph database engines, e.g., random test generation, the most commonly adopted approach in practice. However, random test generation faces the challenge of generating complex inputs (i.e., property graphs and queries) for producing non-empty query results; generating such type of inputs is important especially for detecting wrong-result bugs. To address this challenge, in this paper, we propose GDsmith, the first approach for testing Cypher graph database engines. GDsmith ensures that each randomly generated query satisfies the semantic requirements. To increase the probability of producing complex queries that return non-empty results, GDsmith includes two new techniques: graph-guided generation of complex pattern combinations and data-guided generation of complex conditions. Our evaluation results demonstrate that GDsmith is effective and efficient for producing complex queries that return non-empty results for bug detection, and substantially outperforms the baselines. GDsmith successfully detects 28 bugs on the released versions of three highly popular open-source graph database engines and receives positive feedback from their developers.

CCS CONCEPTS

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

*Tao Xie is the corresponding author.

KEYWORDS

Graph database systems; Differential testing; Cypher

ACM Reference Format:
Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, Wenpin Jiao, and Tao Xie. 2023. GDsmith: Detecting Bugs in Cypher Graph Database Engines. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598046

1 INTRODUCTION

In recent years, graph database engines have been widely used in database applications from various domains, such as knowledge reasoning systems [26] and recommender systems [12, 23]. Graph database engines use the model of labeled property graph [28] (in short as property graph) or the Resource Description Framework (RDF) graph model [1] to represent and store data. As a well-known representative of graph database engines, Neo4j [15] has long been ranked the first in the DB-Engine Ranking [25] (which ranks graph database engines monthly based on their popularity). Over 800 enterprise customers, including over 75% of Fortune 100 companies [25], use Neo4j. Although there is currently no query language standard for graph database engines, it is generally believed that Cypher [8, 9], originally contributed by Neo4j, is the most widely adopted query language specially designed for graph database engines because of Neo4j's overwhelming market share [16]. As an open query language, Cypher is now used by over 10 other graph database engines (e.g., RedisGraph [17] and Memgraph [14]) and tens of thousands of developers [16]. Some graph database engines that natively support other graph query languages (e.g., Gremlin [7]) are also compatible with Cypher queries via translation tools (e.g., Cypher for Gremlin [6]).

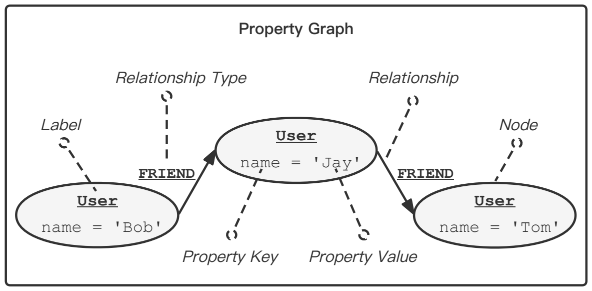
Like other software systems, graph database engines contain bugs, falling into two main categories. (1) Crash bugs. Crash bugs can cause uncaught exceptions to be thrown during query execution. When these bugs are triggered, the users of the database engines are usually informed by error messages given by the database engines. (2) Wrong-result bugs. Wrong-result bugs can cause incorrect return value of queries to be generated during or after query execution. Compared to crash bugs, wrong-result bugs can happen silently without any message given to the users.

To detect bugs in Cypher graph database engines, one can generate test inputs consisting of two parts shown in Figure 1. (1) A **property graph** (which stores data in a structured way) is a directed graph consisting of labeled entities (i.e., *nodes* and *relationships*)[1] and *properties* on these entities. A relationship encodes a directed connection between exactly two nodes. A property is a pair consisting of a *property key* and a *property value* (which is an instantiation of one of Cypher's types such as `STRING`). (2) **Cypher queries** are built up using various clauses for querying the property graph by subgraph pattern matching and data manipulation [16]. In each Cypher query, clauses are chained together, and a clause feeds an intermediate result table to the next clause in the chain. For example, the `MATCH` clause in Figure 1 retrieves all subgraphs (in the property graph) that match the pattern `(user:User)-[r1:FRIEND]-()-[r2:FRIEND]-(fof)`. The retrieved data is then passed to the next `WHERE` clause as a table of matching results. The `WHERE` clause then manipulates the received table by filtering away all lines (in the table) that do not satisfy the condition `user.name = 'Bob'`. The last part of the Cypher query is a `RETURN` clause, which returns the final data to the users.

When test generation is conducted for graph database engines, it is important to generate complex inputs (property graphs and queries) for producing non-empty query results, especially when aiming to detect wrong-result bugs, for two main reasons. First, a high ratio of queries returning empty results reduce the ability for finding wrong-result bugs. Specifically, exposing a wrong-result bug based on a functional-correctness oracle (e.g., differential oracle [13] and metamorphic oracle [4]) requires wrong internal states to be propagated to a query's return value. However, empty-result queries stop the propagation chain in its last phase and result in a trivial empty result. Second, testing effectiveness can be enhanced with automatically generated complex test inputs (property graphs and queries). Graph database engines fall into the type of software systems that have inputs with semantic richness. For this type of systems, complex inputs can explore atypical combinations of features or atypical code that is important but underrepresented in manually written test suites [30].

However, it is challenging for the commonly used random test generation approaches [10, 24, 31], which generate property graphs and queries separately without prior knowledge of each other, to generate a high ratio of non-empty-result queries for two main reasons. First, randomly generated pattern combinations in queries are hard to be matched in a randomly generated property graph. For example, the pattern combination
`(n0:L1)-[r1:T0]->(n2)<-[r3:T1]-(n4:L2), (n2)<-[r4:T2]-(n5:L2)`
(fetched from a minimized query that triggers a wrong-result bug of RedisGraph [17]) describes a subgraph consisting of 4 nodes and 3 relationships with labels and relationship types. Such a complex pattern combination is hard to be matched in a randomly generated property graph. Second, complex conditions can easily become unsatisfiable for a property graph. A randomly generated complex

[1]A node may be assigned with a set of unique *labels*, whereas a relationship is assigned with exactly one *relationship type*.



Cypher Query

```
MATCH (user:User)-[r1:FRIEND]-()-[r2:FRIEND]-(fof)
WHERE user.name = 'Bob' RETURN fof.name AS fofName;
```

Figure 1: An example property graph (containing three `User` nodes and two `FRIEND` relationships) and a Cypher query (finding friends of friends of `Bob`).

condition can be perpetually false if it contains contradictory subconditions such as `x > 0 and x < 0`. Even if the value of a condition with variables can be true given a specific set of variable values (e.g., for condition `x > 0 and x < 2`, if we assign x to 1, the value of the condition is true), it is still possible that the property graph contains no such set of values.

To address the aforementioned challenge, in this paper, we propose GDsmith, the first automated approach, consisting of two main techniques, for testing Cypher graph database engines. (1) **Graph-guided generation of complex pattern combinations**. To generate complex pattern combinations while preserving a high ratio of non-empty-result queries, GDsmith uses the graph information as guidance for pattern generation in queries. Specifically, we record a graph that we generate and extract patterns from it. Then we mutate the extracted patterns and use them for query generation. (2) **Data-guided generation of complex conditions**. Conditions are the root boolean expressions in `WHERE` clauses. To generate complex satisfiable query conditions, GDsmith conducts static query analysis during the generation process of queries. Specifically, GDsmith maintains a value table that records the value of each query variable at each generation point. GDsmith then uses constraints for expression generation to ensure that the value of each condition expression is true for the recorded variable values (meaning that the recorded variable values are not filtered away by any `WHERE` clause).

We implement GDsmith to detect bugs in three highly popular graph database engines (Neo4j [15], RedisGraph [17], and Memgraph [14]), detecting **28 bugs** on their released versions. Among the 28 detected bugs, 23 are confirmed by the developers of the corresponding engines and 14 are already fixed. The developers of all the three graph database engines have replied that our work contributes to their development. The positive feedback from the developers also shows GDsmith's high value in practice. We further evaluate the effectiveness of GDsmith's non-empty-result strategy. Our experimental results show that compared to the baseline (which adopts random test generation), GDsmith detects 50% more failures in 12 hours and achieves a higher non-empty-result ratio.

This paper makes the following main contributions:

- **GDsmith**. The first approach of automated test generation for detecting bugs in Cypher graph database engines.
- **Techniques**. Two techniques for generating complex inputs, achieving a high ratio of non-empty-result queries and increasing the efficiency in finding wrong-result bugs.
- **Evaluations**. Evaluation results for demonstrating GDsmith's effectiveness (substantially outperforming the baseline) and practicability (successfully detecting 28 bugs).
- **Implementation**. GDsmith's tool source code being publicly available at https://github.com/ddaa2000/GDsmith.

2 BACKGROUND AND MOTIVATION

In this section, we first introduce the graph model and query language used in Cypher graph database engines. Then we illustrate the motivation for our work.

2.1 Graph Model in Cypher Databases

Cypher graph database engines adopt the **Property Graph Model** for graph storage and query execution. All data in a Cypher graph database engine is represented and stored as **property graphs**. A property graph is defined as a directed, vertex-labeled, edge-labeled multigraph with self-edges [16]. The Property Graph Model leverages the term **node** to denote a vertex and the term **relationship** to denote an edge. Each node contains a set of unique **labels** (tokens that are used to describe the type of a node) while each relationship contains exactly one **relationship type** (a token that is used to describe the type of a relationship). Both nodes and relationships are **entities** in the Property Graph Model and can hold a set of **properties** (key-value pairs that hold the data of node or relationship entities).

To further understand the aforementioned concepts, we take the property graph in Figure 1 as an example. The property graph contains three different nodes with the same label and two different relationships with the same relationship type. Each node in the property graph contains a label `User`, which indicates that each node represents an instance of the `User` entity. A relationship with a relationship type `FRIEND` from node A to node B represents that `User` A is a friend of `User` B. Each node with a `User` label also contains a string property `name`, which stores the name of the corresponding `User` instance.

This example can also be described by the Entity-Relationship Model [3] (E-R model) and stored by relational tables. Specifically, we can define an E-R model with one E-R entity `User` that contains a property `name` and an E-R relationship `FRIEND`[2]. However, for some complicated situations, the E-R model and tables in relational databases cannot elegantly represent equivalent data structures described by the Property Graph Model. Specifically, the Property Graph Model is a schema-less data model. Each node in a property graph can be assigned with zero to more than one label. The labels and properties of a node are decoupled, meaning that each node with any set of labels can contain any set of properties. Any number of relationships with any kind of relationship type are allowed to connect between two arbitrary nodes.

The schema-less design of the Property Graph Model significantly increases the complexity and flexibility of data organization in graph database engines compared with relational databases. In relational databases, data is represented and stored as table records. The records in the same table are homogeneous. Such records have the same set of properties and use the same set of foreign keys as the reference to records in other tables. However, in graph database engines using the Property Graph Model, nodes with the same set of labels can be heterogeneous. Such nodes can have different sets of properties and may connect to relationships with different relationship types. Therefore, tools for testing relational databases are unable to fully utilize the features of the Property Graph Model. In addition, such complexity and flexibility increase the difficulty of generating non-empty queries. We further illustrate such difficulty in Section 2.2.

[2]As the terms "entity" and "relationship" are used in both the E-R model and the Property Graph Model, we use E-R entity to represent a entity in the E-R model and E-R relationship to represent a relationship in the E-R model.

2.2 The Cypher Query Language

The Cypher query language adopts pattern matching to retrieve subgraphs from a property graph. A **pattern** is a special expression consisting of node and relationship variables (denoted as pattern variables). With a pattern, one can describe a shape of subgraphs satisfying some constraints, which are defined over labels, relationship types, and property values. For example, the pattern `(n0:L1)-[r0]->(n1 {name:'Bob'})` contains two node variables `n0`, `n1` and a relationship variable `r0`. It describes a subgraph with two nodes `n0`, `n1` and one relationship `r0` from `n0` to `n1` where `n0` contains label `L1` and `n1` contains a property `name` with value `'Bob'`.

One basic Cypher query is composed of a sequence of clauses (some complex Cypher queries may contain multiple sequences of clauses) and the clauses execute sequentially. Each clause takes the property graph and a table of intermediate results from the previous clause as inputs and outputs a new table of intermediate results to the next clause. Each column of a table represents one variable of the query and each line of the table represents one set of values for each variable.

A `MATCH` clause is used for pattern matching in Cypher. A `MATCH` clause contains a tuple of patterns that jointly describe a subgraph. It retrieves all the subgraphs that match the pattern tuple from the property graph in a graph database, and passes the retrieved subgraphs as a table (each node or relationship variable forms a column and each subgraph is represented as a line) to the next clause. For example, the following `MATCH` clause contains a pattern tuple with two patterns:

```
MATCH (n0:L1)-[r0]->(n1:L2), (n0)-[r1]->(n2:L3)
```

The preceding `MATCH` clause describes a subgraph consisting of three nodes (`n0`, `n1`, and `n2`) and two relationships (`r0` and `r1`). Cypher allows one node variable to appear multiple times in the patterns of a query and regards them as the same subgraph node. In this example, the node variable `n0` is shared in both patterns so the `MATCH` clause refers to a subgraph where a node `n0` with label `L1` is connected to two nodes n1 and n2 through two relationships r0 and r1, respectively.

For multiple MATCH clauses in the same query, the subgraphs are matched one by one and the results are joined using a subset of their Cartesian product based on their shared variables. For example, the following clause sequence contains two clauses:

```
MATCH  (n0:L1)-[r0]->(n1:L2),  (n0)-[r1]->(n2:L3)
MATCH (n1)-[r2]->(n3:L4)
```

The first clause retrieves the same table of subgraphs matched by the previous example. Then, the second clause gets a table of subgraphs matched by the pattern (n1)-[r2]->(n3:L4). Then, similar to the inner-join operation of SQL, the two tables are joined based on n1 because n1 is the pattern variable that they both share.

In addition to pattern matching, Cypher leverages some other clauses to manipulate data. The most commonly used two clauses are WHERE and WITH clauses. A WHERE clause uses a Boolean expression as the condition to filter the lines in the table. For example, a WHERE clause WHERE n0.name = 'Bob' checks every line in the table and returns only the lines where the value of column n0 contains a property name that has the value being equal to 'Bob'. A WITH clause is used to map the columns of the table to a new set of columns. For example, a WITH clause WITH n0.x * n0.x as square maps a table with column n0 to a table with one column square. Each line in the new table contains a value square that is equal to the square value of n0.x in a line in the previous table.

Pattern-matching operations of Cypher queries have a similar role to table-join operations in SQL queries, especially table-joining operations based on keys. Both of the two kinds of operations are designed to combine sets of homogeneous data. However, due to the complexity and flexibility of the data representation of the Property Graph Model, the space of valid property graphs and valid pattern combinations are relatively large compared with tables. Therefore, given the same limit of data size, the distribution of homogeneous data (e.g., subgraphs that can be matched by the same pattern) in randomly generated property graphs can be sparse, reducing the probability of generating non-empty-result queries. Although one can add constraints for property graphs and queries, such constraints reduce the expressiveness of property graphs and Cypher queries.

2.3 Differential Testing of Graph Database Engines

Differential testing [13] is a widely adopted technique in software testing. The idea of differential testing is that, if a single specification has multiple deterministic implementations, all of these implementations should produce the same output when given the same valid input. Consequently, if different implementations produce two or more distinct outputs, at least one of the implementations violates the specification.

When leveraging differential testing to test Cypher graph database engines, at least two instances should be provided. The instances used for output comparison can belong to different engines, the same engine with different versions, or different configurations. The inputs of a graph database engine can be defined as a property graph and a query to manipulate the graph. The outputs include the property graph after query execution and the results of the query. To meet the principles of differential testing, the inputs provided to database engines should be deterministic, valid, and identical. Therefore, the queries should be syntactically and semantically valid and contain only deterministic features with no undefined behaviors of the Cypher Query Language specification.

2.4 Motivation

To detect bugs in graph database engines, one can generate property graphs and queries as test inputs and use test oracles to check the behavior of graph database engines.

For exception and crash bugs, one can easily detect them by catching the exceptions of query execution and monitoring crash messages of databases. For wrong-result bugs, it is not easy to get the ground-truth execution result of a query because of the rich semantics of the database query language. Instead, one can adopt functional-correctness oracles such as cross-database differential oracles [13] or metamorphic oracles [4] to indirectly search for wrong results based on the relationship between different inputs.

However, even if an error (i.e., wrong inner state) occurs during the execution of a query, the error does not necessarily produce a failure that can be detected by a functional-correctness oracle. The reason is that a wrong inner state in the database engine may not be propagated to the result of the query. Empty-result queries can easily cause such cases. First, empty-result queries usually have empty intermediate data, which can easily eliminate the effect of later data manipulation operations in database queries. For example, for the Cypher query MATCH (n0) WITH n0.val1 * 10 - n0.val2 as a0 RETURN a0, if the property graph is empty (which makes the query return an empty result), the MATCH clause in the query produces an empty table (intermediate data). Therefore, the effect of data manipulation operations in the WITH clause is eliminated because there is no node that can be matched by n0. Second, empty-result queries can erase wrong intermediate data. For example, for the Cypher query MATCH (n0) WITH n0.val1 * 10 - n0.val2 as a0 MATCH (n0)-[r]->(n1) RETURN a0, if the property graph contains no relationship (which makes the query return an empty result) and the WITH clause produces a wrong value of a0, the second MATCH clause maps the result to an empty table, and the wrong value is not propagated to the RETURN clause.

Generating complex test inputs is a strategy widely used for databases and other complex software systems that have inputs with semantic richness (e.g., compilers) [30]. One possible reason for the effectiveness of this strategy is that complex inputs can explore atypical combinations of input features in such software system and cover atypical paths that are not unimportant but underrepresented in manually written test suites of the software system [30].

However, it is challenging to generate complex test inputs while preserving a high non-empty-result query ratio for graph database engines. For graph database engines, the test inputs include property graphs and queries for manipulating the property graphs. More complex queries usually contain more constraints (over property graphs) that are hard to be satisfied. Therefore, generating property graphs and queries separately without prior knowledge of each other can easily lead to a low non-empty-result query ratio.

Algorithm 1 The top-level algorithm of GDsmith

Input: D_a: one instance of a graph database engine; D_b: the other instance of a graph database engine; N_g: the maximum number of newly generated queries.
Output: B: bug reports.

```
1:  while the timeout does not exceed do
2:      $S \leftarrow$ GeneratePropertyGraphSchema($D_a, D_b$)
3:      $G \leftarrow$ GeneratePropertyGraph($S, D_a, D_b$)
4:      $B \leftarrow \emptyset$
5:      $i \leftarrow 0$
6:      while $i < N_g$ do
7:          $Q \leftarrow$ GenerateCypherQuery($S, G$)
8:          $R_a \leftarrow$ ExecuteCypherQuery($Q, D_a$)
9:          $R_b \leftarrow$ ExecuteCypherQuery($Q, D_b$)
10:         if $R_a \neq R_b$ or an exception is caught then
11:             $B \leftarrow B \cup < G, Q >$
12:         end if
13:         $i \leftarrow i + 1$
14:     end while
15: end while
16: return $B$
```

For graph database engines, the large input space of valid property graphs and valid pattern combinations makes it even harder to generate non-empty-result queries compared with relational databases.

3 APPROACH

We propose GDsmith, the first automated approach for testing Cypher graph database engines. GDsmith is black-box, portable, and compatible without any requirement of code instrumentation. Given multiple different instances of graph database engines under test, GDsmith automatically outputs test inputs, each of which includes both a property graph and multiple Cypher queries. GDsmith leverages differential testing [13] to detect whether a bug is triggered (with a test oracle) regardless of crashing or not. For example, GDsmith users can use cross-engine (i.e., comparing results fetched by different graph database engines), cross-version (i.e., comparing results fetched by different versions of the same graph database engine), or cross-optimization (i.e., comparing results fetched by different query options on the same graph database engine) differential oracles.

The overall algorithm of GDsmith is shown in Algorithm 1. In particular, GDsmith generates test inputs through a four-step iteration. First, GDsmith randomly generates a database schema that defines the set of labels and properties in the property graph (Line 2, shown in Section 3.1). Second, GDsmith randomly generates a property graph and feeds it into the graph database engine instances under test (Line 3, shown in Section 3.1). GDsmith also stores the property graph for later query generation. Third, GDsmith generates semantically valid Cypher queries based on the guidance of the property graph (Line 7, shown in Section 3.2). GDsmith ensures that each query is deterministic, valid, and has no undefined behaviors; thus, all properly implemented graph database engine instances should return the same results. Fourth, GDsmith executes the generated queries on each engine instance and records the results returned by each engine instance (Lines 8 and 9). In the end, if a discrepancy between results is found or an exception of any graph database engine is caught, GDsmith generates a bug report including the property graph and Cypher queries that trigger the bug (Lines 10-12, shown in Section 3.3). In the rest of this section, we illustrate the details of these steps.

3.1 Schema and Property Graph Generation

In each iteration, GDsmith first randomly generates a property graph schema. A property graph schema defines the labels, the relationship types, and the properties on the labels and the relationship types [11]. In particular, GDsmith randomly generates a set of labels and a set of relationship types (the maximum size of each set is predefined by the GDsmith users). Each label or relationship type has a unique name and a set of associated properties[3]. GDsmith generates a unique name and a data type for each property. Additionally, GDsmith randomly generates a set of constant values as the possible values for each property.

GDsmith then randomly generates a property graph and feeds it into each graph database engine instance under test. Specifically, GDsmith first generates a set of nodes with labels and properties. Each node is randomly given zero to many labels. GDsmith then randomly adds properties to the nodes. Note that each node contains only a subset of the properties that are determined by its labels. GDsmith randomly generates relationships for the property graph by randomly selecting node pairs and adding a relationship between the nodes in each pair. GDsmith finally adds exactly one relationship type and multiple properties to each relationship using the same way as node generation.

3.2 Query Generation

GDsmith includes a grammar-based generator for query generation. The goal of the generator is to generate a set of semantically valid queries with a high non-empty result ratio. To ensure the semantic validity of queries, GDsmith uses a set of built-in rules to ensure that the generated queries are both syntactically correct and semantically valid. To increase the non-empty result ratio of queries, for the generation of patterns, GDsmith uses the graph recorded in the graph generation step to guide the generation of patterns. For the generation of conditions, GDsmith conducts static query analysis during the generation process to generate conditions that can be satisfied by the data in the property graph.

3.2.1 Overview of the Query Generation Process. GDsmith includes a two-step technique to generate a query. Figure 2 outlines the generation process. First, GDsmith randomly generates a Cypher skeleton. We refer to a clause sequence with uninstantiated parts (e.g., patterns and expressions) as a Cypher skeleton. Specifically, a Cypher skeleton can be considered as a language $\mathcal{L}_{skeleton}$ generated by the grammar shown in Table 1. The grammar of Cypher skeletons is consistent with the Cypher syntax except that each uninstantiated part p is denoted as $\bigcirc_p$.

[3]Although the Property Graph Model is a schema-less data model, and the labels and relationship types are decoupled with the properties, the implementation of some graph databases still requires a pre-defined schema where the properties of each label or relationship type are defined using special schema building queries.

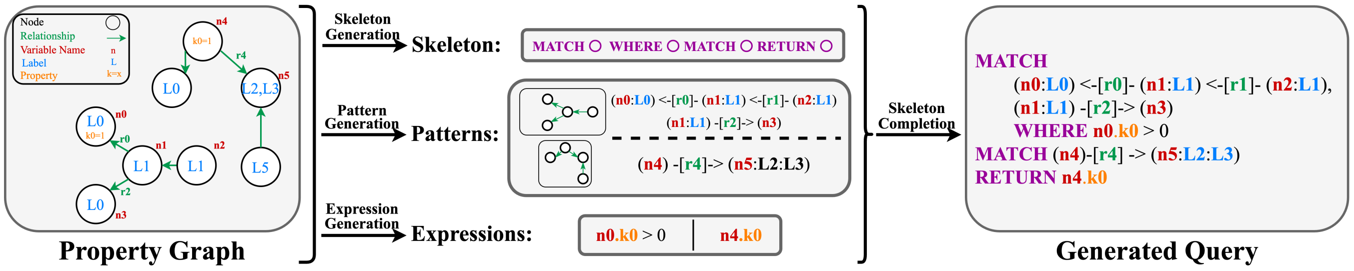


Figure 2: The overview of query generation

Table 1: Core grammar of Cypher skeletons

skeleton	::=	`RETURN` *ret* [`ORDER BY` {$\bigcirc_o$}] [`SKIP` $\bigcirc_s$] [`LIMIT` $\bigcirc_l$] | *clause skeleton*
ret	::=	$*$ | {$\bigcirc_r$ [`AS` $\bigcirc_a$]}
clause	::=	[`OPTIONAL`] `MATCH` {$\bigcirc_m$} [`WHERE` $\bigcirc_w$] | `WITH` *ret* [`WHERE` $\bigcirc_w$] [`ORDER BY` {$\bigcirc_o$}] [`SKIP` $\bigcirc_s$] [`LIMIT` $\bigcirc_l$]
		| `UNWIND` $\bigcirc_u$ [`AS` $\bigcirc_a$]

Algorithm 2 The GenPatterns function

```
Input: $G$: the property graph.
Output: $Patterns$: the patterns that can be used to complete skeletons; $Vals$: the initial values of entity variables.
1:  function GenPatterns($G$)
2:      $i \leftarrow 0$
3:      $Patterns \leftarrow \emptyset$
4:      $Vals \leftarrow \emptyset$
5:      while $i < MaxRegionNumber$ do
6:          $region$ ← Randomly get one region (connected subgraph) from $G$
7:          $m \leftarrow$ NewVars($region$)
8:          $Vals \leftarrow$ GetVals($m$) $\cup$ $Vals$
9:          $j \leftarrow 0$
10:         while $j < MaxPatternInRegion$ do
11:             $g \leftarrow$ ExtractSimpleSubgraph($region$)
12:             $p \leftarrow$ TranslateToPattern($g, m$)
13:             $Patterns \leftarrow Patterns \cup \{p\}$
14:             $j \leftarrow j + 1$
15:         end while
16:         $i \leftarrow i + 1$
17:     end while
18:     return $Patterns, Vals$
19: end function
```

Second, GDsmith completes the patterns and expressions in the skeleton according to the semantics of Cypher. To ensure the semantic validity of queries, GDsmith uses a set of built-in rules over value types and variable life cycles during generation. To increase the non-empty result ratio of queries, for the generation of patterns, GDsmith uses the graph recorded in the graph generation step to guide the generation of patterns while for the generation of expressions, GDsmith conducts static query analysis during the generation process to generate condition expressions that are satisfiable.

3.2.2 *Graph-guided Pattern Generation.*

In this subsection, we describe the pattern generation algorithm of GDsmith. The main goal of this algorithm is to generate pattern combinations that match at least one set of subgraphs in the generated property graph. In addition, we try to generate diverse pattern combinations that have both patterns with shared entity variables (such patterns together describe a complex connected subgraph) and patterns with isolated entity variables (such patterns describe subgraphs unrelated to each other).

To achieve our goal, the conceptual idea of GDsmith is to select multiple self-connected subgraphs (we refer to such self-connected subgraphs as regions), and construct patterns by mapping the nodes and relationships into entity variables. The sets of entity variables between regions are disjointed so only the patterns constructed from the same region share common entity variables.

Algorithm 2 outlines the process of pattern generation. In each iteration (Lines 5-17), GDsmith randomly extracts a region from the property graph G (Line 6). GDsmith then creates a new entity variable for each node or relationship in the region, and uses m to record the mapping from these entities to entity variables (Line 7). GDsmith then randomly extracts simple subgraphs from the region (Line 11). We refer to subgraphs that can be directly represented by one Cypher pattern as simple subgraphs. GDsmith then translates each simple subgraph to a pattern p (Line 12). Specifically, the structure of the simple subgraph is represented with the corresponding entity variables in m. The labels and relationship types of the nodes and relationships in the subgraph are added as constraints in the pattern.

Figure 3 gives an example of translating simple subgraph g to pattern p. Suppose that g has two nodes `N0` and `N1`, and one relationship `R0` from `N0` to `N1`, and m records that entity variables `n0`, `n2`, and `r5` are used to represent `N0`, `N1`, and `R0`, respectively. The function *TranslateToPattern* first reads g to determine that the structure of g should be represented by a pattern with the structure `(?)-[?]->(?)` or `(?)<-[?]-(?)`. Suppose that the first pattern structure is selected, the function then fills in the pattern with entity variables `n0`, `r5`, and `n2`, and gets `(n0)-[r5]->(n2)`. Finally, the

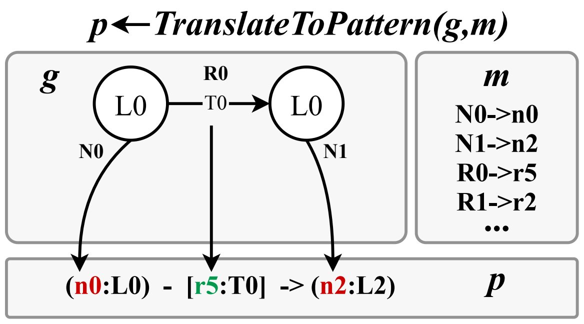


Figure 3: Example of `TranslateToPattern`

Algorithm 3 The GenExpr function

Input: c: a constraint that the generated expression needs to satisfy; $Vals$: the value table of variables; $type$: the expected value type of the expression.

Output: e: the expression tree that satisfies c and has the type $type$.

```
1:  function GenExpr($c, Vals, type$)
2:      $e \leftarrow$ Get one expression skeleton that has $type$
3:      if $e$ requires identifiers then
4:          $e.consts \leftarrow$ GenConsts($e, c$)
5:          $e.vars \leftarrow$ GenVarRefs($e, c, Vals$)
6:          $e.funcs \leftarrow$ GenFuncRefs($e, c$)
7:      end if
8:      $childPosCons \leftarrow$ SplitConstraints($e, c$)
9:      for each $p, cc \in childPosCons$ do
10:         $e.child[p] \leftarrow$ GenExpr($cc, Vals$, Type($e.child[p]$))
11:     end for
12:     return $e$
13: end function
```

function selects some labels and relationship types as constraints and adds them to the pattern, and gets `(n0:L0)-[r5:T0]->(n2:L2)`.

3.2.3 Data-guided Condition Generation. To generate queries with satisfiable conditions, GDsmith conducts static query analysis to guide the generation of conditions. Here, a condition in a query is satisfiable if at least one group of data in the property graph can be retrieved and mapped to a line of the intermediate table produced by the corresponding `WHERE` clause. Take the following query as an example:

```
MATCH (n0)-[r0]->(n1) WHERE n0.k0 = 'x' WITH n0.k1
+ n1.k1 as a0 WHERE a0 > 5000 RETURN *
```

The conditions `n0.k0 = 'x'` and `a0 > 5000` are satisfiable if there exists one subgraph `(n0)-[r0]->(n1)` where `n0` has a property `k0` that equals `'x'` while both `n0` and `n1` have a property `k1` and the sum of them is greater than 5000.

To achieve the goal, GDsmith maintains a value table $Vals$ for each query during the generation process. The value table is a mapping where its keys are query variables and the value of a key is the value of the variable at the present generation point. The data-guided condition generation can be divided into the following three parts: value table initialization, value table maintenance, and condition generation.

Value table initialization. The value table is initialized in the pattern generation process. Specifically, when a new entity variable is created according to an entity, its value is recorded exactly as the entity with its labels, relationship type, and property values (Lines 7-8 in Algorithm 2). Note that each entity variable can possibly match other entities from the property graph, but the strategy of GDsmith is to record only one value.

Value table maintenance. After the filling of each clause, GDsmith updates the value table to maintain the values of variables at the generation point. For each variable defined or updated in the present clause, GDsmith adds or updates the variable and its value in the value table. The value of the variable is calculated by simulating the expression assigned to the variable. For some complicated built-in functions (e.g., the SUM aggregation function) the value is hard to predict without execution so GDsmith assigns the special value $UNKNOWN$ to the expression. Any expression that contains a sub-expression with the value $UNKNOWN$ will also be assigned to the value $UNKNOWN$.

Condition generation. Algorithm 3 outlines the process of expression generation with constraints. This algorithm takes a parameter c that describes a constraint and a parameter $type$ that specifies the value type requirement for the generated expression. This algorithm also takes the value table $Vals$ for information on available variables and their values. Generally, this algorithm generates an expression that satisfies c and has the value type of $type$. GDsmith uses this algorithm to generate all expressions in a Cypher query. For condition expressions in `WHERE` clauses, the parameter c is set to $value = true$, which indicates that the value of the expression should equal $true$. For other expressions, c is simply assigned with nil, which indicates that there is no constraint for the value of the expression.

GDsmith uses a top-down technique to generate an expression tree. GDsmith first randomly selects a Cypher expression skeleton e having the type $type$ (Line 2). Then GDsmith completes e with identifiers and sub-expressions. If e requires no sub-expression, GDsmith generates its identifiers (constants, variables, and functions) to satisfy c (Lines 3-7). For example, if e is a variable reference expression and c is $value > 100$, GDsmith selects from $Vals$ an integer variable or variable property that has a value greater than 100.

If e requires sub-expressions, GDsmith splits c into a set of sub-constraints required for each sub-expression and ensures that if all sub-expressions satisfy their corresponding sub-constraints, c will be satisfied (Line 8). To achieve this goal, GDsmith uses a set of designed rules for each kind of expression to split the constraint. For example, for an "or" expression, if c is $value = true$, to satisfy c, at least one sub-expression of the "or" expression should have the value $true$. Therefore, GDsmith splits c to the constraint for left sub-expression nil and the constraint for right sub-expression $value = true$. Finally, GDsmith recursively generates all sub-expressions with sub-constraints and types, and constructs the final expression by filling in e with its generated sub-expressions (Lines 9-11). One special case is an expression with the value $UNKNOWN$. As we cannot calculate the value of such expressions, all such expressions are considered to satisfy the constraint.

3.3 Bug Detection

By comparing the returned results of the same Cypher query on different instances of graph database engines, GDsmith can detect two main types of bugs. (1) A **crash bug** is triggered if a syntactically correct and semantically valid query cannot be successfully executed (e.g., throwing an exception). Such bugs prevent users from obtaining expected results, and in more serious cases, cause the graph database engine under test to crash and lose connection to upper-level database applications. (2) A **wrong-result bug** is triggered if a syntactically correct and semantically valid query is executed successfully but returns incorrect results. Such bugs are more dangerous because users may mistakenly believe that the Cypher query returns correct results and have wrong expectations about the behavior, leading to potential risks.

Note that when GDsmith aims to detect wrong-result bugs, there are prerequisites for each Cypher query. First, the Cypher query containing non-deterministic sub-clauses may result in false alarms. For example, the results will get trimmed from the top by using the `SKIP` sub-clause. However, without an `ORDER BY` sub-clause, the records are randomly selected because no guarantees are made on the order of the results [16]. To avoid such false alarms, GDsmith uses deterministic clauses and routines for query generation. Second, undefined behaviors make it hard to determine whether inconsistent results are bugs or just different implementations. For example, integer overflows and divisions by zero are handled differently by different engines (e.g., returning `NaN` or throwing exceptions). GDsmith does not generate Cypher queries that may exhibit such undefined behaviors. Third, for convenient and efficient comparison, a Cypher query should return only a few specific expressions instead of a large number of entities. For example, when executing a Cypher query whose `RETURN` clause contains the $*$ symbol on each graph database engine, parsing and comparing results including all nodes and relationships from different engine instances are time-consuming because an entity may consist of plenty of property values.

4 EVALUATIONS

In our evaluations, we address the following three research questions (RQs):

- **RQ1:** How effectively can GDsmith generate test inputs compared to the baselines?
- **RQ2:** How much do different techniques in GDsmith contribute to the overall effectiveness of GDsmith?
- **RQ3:** How practicably does GDsmith detect real-world bugs in popular graph database engines?

4.1 Evaluation Setup

4.1.1 Subjects. We select three popular real-world graph database engines as our evaluation subjects. (1) Neo4j [15] is the market leader, graph database category creator, and the most widely deployed graph data platform in the market. It has long been ranked first in the DB-Engines Ranking [25]. It is a high-performance graph store with all the features expected of a mature and robust database. (2) RedisGraph [17] is the first queryable property graph database to use sparse matrices to represent the adjacency matrix in graphs and linear algebra to query a graph. (3) Memgraph [14] is a streaming graph application platform and leverages an in-memory-first architecture.

We test the released versions of the Neo4j Community Edition from 3.5 to 4.4, the released versions of RedisGraph 2.8, and the released version 2.4 of the Memgraph Community Edition. All of these subjects are downloaded from their official repositories without any modification.

4.1.2 Implementation. We implement the GDsmith prototype with over 22K non-comment lines of Java code. Its framework is derived from SQLancer [18] (which is a tool to automatically test relational database engines). GDsmith uses Neo4j Java Driver 4.1.1 to connect and interact with Neo4j and Memgraph, and uses JRedisGraph 2.5.1 to connect and interact with RedisGraph. The default length of clauses is set to nine. The default number of nodes in a graph is set to 128. The default maximum number of patterns in each `MATCH` clause is set to four. The default maximum depth of expression trees is set to two.

Some graph database engines implement only a subset of the Cypher language. When conducting cross-engine differential testing, we configure GDsmith in advance so that all Cypher queries generated by GDsmith do not contain any Cypher feature that is unsupported by any of the three graph database engines. All evaluations are conducted on a Ubuntu 20.04 server with two AMD EPYC-7H12 CPUs and 512 GB of memory.

4.1.3 Baselines. There is no applicable baseline to which we can compare our work because no existing work focuses on detecting bugs in Cypher graph database engines and GDsmith is the first approach for this purpose. Therefore, we design and implement two variants of GDsmith for our evaluations to analyze the respective contribution of our techniques. One variant named GDsmith$_{!pc}$ disables both the graph-guided generation of complex pattern combinations and the data-guided generation of complex conditions, and generates semantically valid patterns and conditions randomly. The other variant named GDsmith$_{!c}$ disables only the data-guided generation of complex conditions. Note that there is no variant named GDsmith$_{!p}$ that disables only the graph-guided generation of complex pattern combinations because the data-guided generation of complex conditions relies on the information provided by the graph-guided generation of complex pattern combinations.

4.1.4 Metrics. To measure the effectiveness of GDsmith, we design the following three types of metrics:

(1) The **number of version-distinct discrepancies** (N_{VDD}) is used to estimate the number of distinct bugs detected in the given time period. Consider that we have a set of test cases $\mathbf{t} = \{t_1, t_2, ..., t_k\}$ that trigger discrepancies between a buggy graph database engine instance e_f and a ground-truth instance e_t. Now, we want to estimate how many distinct bugs these discrepancies reflect. We run each test case t_i on a set of additional engine instances $\mathbf{e} = \{e_1, e_2, ...e_n\}$ sharing the same engine with e_f but having different versions. We compare the results of these instances with the result of e_f to check whether they reproduce a discrepancy (we consider

that a discrepancy is reproduced in e_j if the result of e_j is identical to the result of e_f) and get the reproducing version set calculated for the test case: v_i ={e_j | e_j reproduces the discrepancy }. The number of version-distinct discrepancies detected by these test cases is defined as the number of distinct reproducing version sets calculated for these test cases.
This metric is similar to the Correcting Commits measurement, which is widely used to approximate the number of unique bugs detected by compiler testing [2]. We use versions instead of commits to reduce the cost of running each discrepancy triggering test input. Although this metric has a high tendency to underestimate the number of unique bugs as multiple bugs can be fixed or introduced in the same version, we can still use it to measure the number of unique wrong-result bugs for comparison between approaches. A high number of version-distinct discrepancies indicate that the generated inputs can effectively detect distinct wrong-result bugs.

(2) The **number of discrepancies** (N_D) is defined as the number of discrepancies detected within the same time period. A high number of discrepancies indicate that the generated inputs are able to detect wrong-result bugs at a high speed.
(3) The **non-empty-result query ratio** is defined as the percentage of queries returning non-empty results among all generated Cypher queries. A high non-empty-result query ratio indicates that the generated queries are meaningful toward differential testing, especially for detecting wrong-result bugs.

4.2 RQ1/RQ2: Effectiveness

To assess the effectiveness and efficiency of GDsmith and the respective contribution of our techniques, we run GDsmith, GDsmith$_{!c}$, and GDsmith$_{!pc}$ for the same period of time and measure the metrics of these three approaches. Specifically, we conduct two experiments as shown in Table 2. The first experiment uses cross-engine difference as the oracle to assess the effectiveness and efficiency of GDsmith on recent release versions of graph databases. The database engine instances under test are Neo4j Community 4.4.12, RedisGraph Docker 2.8.20, and Memgraph Docker 2.4.0, which are the recent release versions of each engine. The second experiment uses cross-version difference as the oracle. It takes Neo4j Community 3.5.0 and Neo4j 4.4.12 as the engine instances under test. We use Neo4j Community 3.5.0 because old versions of graph database engines usually contain more bugs and can provide more statistically significant results compared to recent versions. We run each approach for 12 hours for each experiment and collect all the detected discrepancies.

To calculate the number of version-distinct discrepancies, for the cross-version experiment, we store all test inputs that trigger discrepancies and re-run them in different versions of Neo4j Community from 3.5.0 to 4.4.20 (all release versions, including maintenance versions). We use version 4.4.20 as the ground-truth instance to determine whether the instance of 3.5.0 or 4.4.12 is buggy and then calculate the number of version-distinct discrepancies. For the cross-engine experiment, we select 10% of the discrepancy-triggering test

Table 2: Experiment groups of GDsmith

Experiment	**Engine Instances**
Cross-engine	Neo4j 4.4.12
	RedisGraph 2.8.20
	Memgraph 2.4.0
Cross-version	Neo4j 4.4.12
	Neo4j 3.5.0

Table 3: Discrepancies detected by the three approaches in cross-database and cross-version experiments. N_D is the number of discrepancies. N_{VDD} is the number of version-distinct discrepancies

Experiment	**Approach**	N_D	N_{VDD}
Cross-engine	GDsmith	11275	26
	GDsmith$_{!c}$	9286	22
	GDsmith$_{!pc}$	8815	17
Cross-version	GDsmith	483	18
	GDsmith$_{!c}$	199	9
	GDsmith$_{!pc}$	87	7

inputs and re-run them in different versions of Neo4j Community from 3.5.0 to 4.4.20, RedisGraph Docker from 2.8.0 to 2.8.26 and Memgraph Docker from 2.0.0 to 2.7.0. We use majority voting of the three newest versions of each engine as the oracle. Only if at least one pair of the newest versions retrieves the same result for a test input, we keep the test input. In addition, we discard the test inputs that trigger crash bugs in the selected versions as these test inputs may introduce different bugs to interfere with the calculation of this metric.

4.2.1 Wrong-result Bug Detection. Table 3 shows the N_D and N_{VDD} values of the three approaches in two experiments. In both experiments, GDsmith gets higher N_D and N_{VDD} than the two baselines. GDsmith$_{!c}$ gets higher N_D and N_{VDD} than GDsmith$_{!pc}$. The results indicate that both techniques of GDsmith contribute to the effectiveness in detecting wrong-result bugs.

Figure 4 shows the N_D of the three approaches over time. In both cross-version and cross-engine experiments, the N_D of GDsmith grows faster than the two baselines. In the cross-version experiment, the N_D of GDsmith$_{!c}$ grows faster than GDsmith$_{!pc}$ in the cross-version experiment. In the cross-engine experiment, the N_D of GDsmith$_{!c}$ and GDsmith$_{!pc}$ have similar growth speeds.

Figure 5 shows the N_{VDD} of the three approaches in the cross-version experiment. The N_{VDD} of GDsmith grows faster than the two baselines, and the N_{VDD} of GDsmith$_{!c}$ grows faster than GDsmith$_{!pc}$ in the cross-version experiment.

The evaluation results show that GDsmith is more effective in detecting discrepancies than the two baselines. The results of GDsmith$_{!pc}$ and GDsmith$_{!c}$ also show that both of our techniques contribute to the overall effectiveness of GDsmith.

4.2.2 Non-empty-result Query Ratio. We run the three approaches for 1 hour. Table 4 shows the non-empty-result query ratio of each approach. GDsmith achieves the non-empty-result query ratio of 73.66% while GDsmith$_{!pc}$ and GDsmith$_{!c}$ achieve 18.19% and 36.79%,

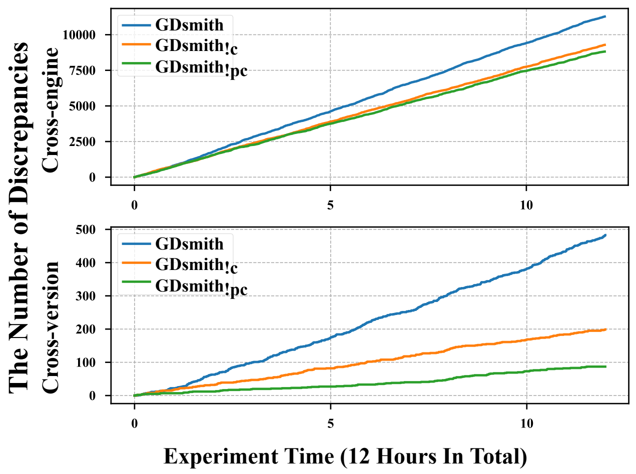


Figure 4: The number of discrepancies detected by GDsmith and two baselines

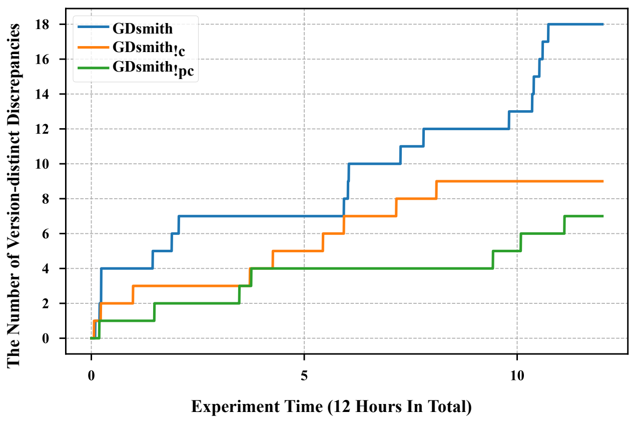


Figure 5: The number of version-distinct discrepancies detected by GDsmith and two baselines in the cross-version experiment

respectively. The results illustrate that the techniques in GDsmith can generate test inputs for increasing the non-empty-result query ratio. Specifically, by comparing the results between GDsmith$_{!pc}$ and GDsmith$_{!c}$, and the results between GDsmith$_{!c}$ and GDsmith, we find that both of our techniques contribute to increasing the non-empty-result query ratio. Note that each technique focuses on one pre-condition for generating non-empty-result queries, which can be generated only when both pre-conditions are satisfied.

4.3 RQ3: Practicability

4.3.1 Study Findings. Table 5 shows, for each subject, the number of crash bugs (Bug$_{cr}$) and wrong-result bugs (Bug$_{wr}$) detected in the second and third columns, respectively, and the number of fixed bugs, confirmed but unfixed bugs, and reported but unconfirmed bugs in the fourth, fifth, and sixth columns, respectively. GDsmith detects 28 bugs in total. Note that all these bugs are detected on the released versions of the engines under test. Among the 28 detected bugs, 23 are confirmed by developers of the corresponding engines and 14 are already fixed.

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

Approach	Non-empty-result Query Ratio
GDsmith	73.66%
GDsmith$_{!pc}$	18.19%
GDsmith$_{!c}$	36.79%

Table 5: Bug detection results of GDsmith

Subject	Bug$_{er}$	Bug$_{wr}$	Fixed	Confirmed	Reported
Neo4j	5	2	7	0	0
RedisGraph	3	11	6	7	1
Memgraph	0	7	1	2	4
SUM	8	20	14	9	5

4.3.2 Example Bugs. We next show the following three examples of confirmed bugs detected by GDsmith to illustrate what kinds of bugs in the three graph database engines can be detected by GDsmith. For brevity, we show only reduced Cypher queries that demonstrate the underlying core problem, rather than the original queries and property graphs that trigger the bugs.

Example Bug 1: Cypher queries that trigger a wrong-result bug in RedisGraph 2.8.20.

```
① CREATE (n0:T2), (n1)<-[r1:T1{id:2, k51:1}]-(n2);
② MATCH (n7) OPTIONAL MATCH (n0:T2), (n7)<-[r7:T1]
-(n8) RETURN (r7.k51) AS a0;
```

The first query uses a `CREATE` clause to create a property graph with three nodes and one relationship. The second query should return a table of three lines: `{a0:1}`, `{a0:null}`, `{a0:null}`. However, the result of RedisGraph is `{a0:1}`, `{a0:1}`, `{a0:1}`. Note that if `(n0:T2)` in the second query is removed, the query will return the correct result table.

The root cause is that one query operator named `Argument` is mistakenly put at the wrong position in the operator tree. We observe two phenomena in such a bug-triggering case. First, this bug can be triggered only through the combination of language features. Although the pattern `(n0:T2)` in the second query should not affect the result of the query on this specific property graph, it is integral to have this pattern to trigger the wrong planning of the `OPTIONAL MATCH` clause. Second, returning non-empty results is a prerequisite to detecting such a bug. Although such wrong planning happens regardless of the structure of the property graph, only non-empty query results can reveal the different behaviors between wrong and correct planning.

Example Bug 2: Cypher queries that trigger a wrong-result bug in Neo4j 4.2.14.

```
① CREATE (n0), (n1);
② MATCH (n0) OPTIONAL MATCH (n3) WITH n3 OPTIONAL
MATCH (n3), (n5) OPTIONAL MATCH (n3) RETURN 1;
```

The first query uses a `CREATE` clause to create a property graph with two unlabeled nodes. Neo4j 4.2.14 mistakenly doubles each returned line.

The root cause is that the planner mistakenly uses an `AllNodesScan` operator for pattern `n3`, and the operator introduces an unnecessary multiplication of rows. Although the patterns in the query are quite simple, triggering the bug requires the combination of multiple clauses, including `OPTIONAL MATCH` clauses and a `WITH` clause.

Example Bug 3: Cypher queries that cause RedisGraph 2.8.20 to crash.

```
① CREATE (n:N);
② MATCH (n:N) OPTIONAL MATCH (n:Q) RETURN 1;
```

The first query creates a node with one label. Then the second query triggers the crash of the database engine instance under test.

The root cause is that the second query triggers the `Label-Scan` optimization strategy in RedisGraph. However, the optimization is buggy and causes the whole engine instance to crash. Note that the query to trigger this bug is simple but atypical because it operates on only one node but matches it multiple times with different labels. However, such a simple query can cause a severe crash and be potentially triggered on more typical queries in future versions if not detected or fixed.

4.4 Threats to Validity

For external validity, the main threat is that the subjects chosen in our evaluations might not be generalized to other subjects. To reduce the threat, we pick three well-known and open-source graph database engines as representatives. In fact, GDsmith is able to test any graph database engine supporting the Cypher language.

For internal validity, the main threat lies in the implementation of GDsmith. Not all Cypher features are currently supported (e.g., `UNION` clauses). To mitigate the threat, we investigate the covered Cypher language features by multiple graph database engines and refer to the description of core Cypher's syntax and semantics, enabling GDsmith to support commonly used grammars.

5 RELATED WORK

Testing Relational Database Engines. There are various approaches to testing relational database engines. SQLsmith [22] continuously generates syntactically correct SQL queries from the abstract syntax tree (AST) directly, meanwhile detecting whether the relational database engine under test faces crashes. Squirrel [33] combines coverage-based fuzzing and model-based generation. It performs type-based mutations on the defined DSL and optimizes for semantic validity with additional analysis. Ratel [27] is an enterprise-level fuzzer that improves the feedback precision, enhances the robustness of input generation, and performs an on-line investigation on the root cause of bugs with its industry-oriented design.

The aforementioned approaches can detect only crashing bugs in relational database engines. To detect wrong-result bugs, RAGS [24] generates and executes SQL queries in multiple relational database engines, and meanwhile observes differences in the output sets. Any inconsistency among results indicates that at least one relational database engine contains bugs. PQS [21] detects wrong-result bugs by checking whether a specific record is fetched correctly. NoREC [19] detects bugs in a relational database engine by applying a semantic-preserving transformation to a given SQL query to disable the engine's optimizations and addresses PQS' high implementation effort. TLP [20] derives multiple SQL queries that compute a partial result of the initial query. By using a composition operator, the partitions can be combined to yield the same result as the original query; if the result differs, a bug in the relational database engine has been detected. MutaSQL [5] generates test cases by mutating a SQL query over a database instance into a semantically equivalent query mutant, and checks the results returned by the relational database engine under test.

Compared with these approaches, GDsmith includes our skeleton-based completion technique to ensure that each randomly generated Cypher query satisfies the semantic requirements. GDsmith also includes our novel techniques to increase the probability of producing Cypher queries returning non-empty results, and these techniques are designed according to unique features of the Cypher language.

Testing Graph Database Engines. There are various approaches to testing graph database engines. The latest work most related to GDsmith is Grand [32]. Grand is a random differential testing tool for Gremlin-based graph databases. It conducts a model-based approach to generate valid Gremlin queries and then uses differential testing to detect wrong-result bugs. RD^2 [29] is another random differential testing tool that detects wrong-result bugs in graph database engines that adopt the RDF model [1]. Compared with these approaches, GDsmith includes our two techniques (namely graph-guided pattern generation and data-guided condition generation) to increase the non-empty result ratio for generated queries, improving the capability of revealing wrong-result bugs.

6 CONCLUSION

In this paper, we have introduced a new important problem of testing graph database engines to detect wrong-result bugs, and have proposed GDsmith, the first automatic testing approach for detecting bugs in Cypher graph database engines. We have implemented GDsmith and evaluated it against the baselines. The evaluation results demonstrate GDsmith's high effectiveness and efficiency. We have also applied it to test three highly popular open-source graph database engines, successfully detecting **28 bugs** on their released versions and receiving positive feedback from their developers.

7 ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China under Grant No. 62161146003, National Science and Technology Major Project No. 2020AAA0109401, and the Tencent Foundation/XPLORER PRIZE. Lu Zhang, Wenpin Jiao, and Tao Xie are also affiliated with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education China.

REFERENCES

[1] Ibrahim Abdelaziz, Essam Mansour, Mourad Ouzzani, Ashraf Aboulnaga, and Panos Kalnis. 2017. Query Optimizations over Decentralized RDF Graphs. In *Proceedings of the 33rd International Conference on Data Engineering*. 139–142. https://doi.org/10.1109/ICDE.2017.59

[2] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An Empirical Comparison of Compiler Testing Techniques. In *Proceedings of the 38th International Conference on Software Engineering*. 180–190. https://doi.org/10.1145/2884781.2884878

[3] Peter Pin-Shan Chen. 1976. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems* (1976), 9–36. https://doi.org/10.1145/320434.320440

[4] Tsong Yueh Chen, Shing Chi Cheung, and Siu Ming Yiu. 1998. Metamorphic Testing: a New Approach for Generating Next Test Cases. *Technical Report HKUST-CS98-01* (1998).

[5] Xinyue Chen, Chenglong Wang, and Alvin Cheung. 2020. Testing Query Execution Engines with Mutations. In *Proceedings of the 8th International Workshop on Testing Database Systems*. 6:1–6:5. https://doi.org/10.1145/3395032.3395322

[6] The Apache Software Foundation. 2022. Cypher for Gremlin. https://github.com/opencypher/cypher-for-gremlin/tree/master/tinkerpop/cypher-gremlin-server-client

[7] The Apache Software Foundation. 2022. Gremlin Query Language. https://tinkerpop.apache.org/gremlin.html

[8] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Martin Schuster, Petra Selmer, and Andrés Taylor. 2018. Formal Semantics of the Language Cypher. *arXiv preprint arXiv:1802.09984* (2018). http://arxiv.org/abs/1802.09984

[9] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data*. 1433–1445. https://doi.org/10.1145/3183713.3190657

[10] Bogdan Ghit, Nicolás Poggi, Josh Rosen, Reynold Xin, and Peter A. Boncz. 2020. SparkFuzz: Searching Correctness Regressions in Modern Query Engines. In *Proceedings of the 8th International Workshop on Testing Database Systems*. 1:1–1:6. https://doi.org/10.1145/3395032.3395327

[11] Lior Kogan. 2017. V1: A Visual Query Language for Property Graphs. *arXiv preprint arXiv:1710.04470* (2017). http://arxiv.org/abs/1710.04470

[12] 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*. 1–8. https://doi.org/10.1109/UIC-ATC.2017.8397433

[13] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* (1998), 100–107. http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[14] Memgraph. 2022. Memgraph: Frictionless, Innovative, Graph Applications. https://memgraph.com/

[15] Neo4j. 2022. The Fastest Path To Graph Productivity: Neo4j Graph Database. https://neo4j.com/product/neo4j-graph-database/

[16] The openCypher Implementers Group. 2022. Cypher Query Language Reference, Version 9. https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf

[17] RedisGraph. 2022. RedisGraph - a Graph Database Module for Redis. https://oss.redis.com/redisgraph/

[18] Manuel Rigger. 2022. SQLancer: Detecting Logic Bugs in DBMS. https://github.com/sqlancer/sqlancer

[19] 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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1140–1152. https://doi.org/10.1145/3368089.3409710

[20] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. *Proceedings of the ACM on Programming Languages* (2020), 211:1–211:30. https://doi.org/10.1145/3428279

[21] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. 667–682. https://www.usenix.org/conference/osdi20/presentation/rigger

[22] Andreas Seltenreich. 2022. Bug Squashing with SQLsmith. https://github.com/anse1/sqlsmith

[23] 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* (2021), 74. https://doi.org/10.1007/s42979-021-00460-8

[24] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the 24rd International Conference on Very Large Data Bases*. 618–622. http://www.vldb.org/conf/1998/p618.pdf

[25] solid IT gmbh. 2022. DB-Engines Ranking of Graph DBMS. https://db-engines.com/en/ranking/graph+dbms

[26] Jian Wang, Ke Wang, Jing Li, Jianmin Jiang, Yanfei Wang, Jing Mei, and Shaochun Li. 2020. Accelerating Epidemiological Investigation Analysis by Using NLP and Knowledge Reasoning: A Case Study on COVID-19. In *2020 American Medical Informatics Association Annual Symposium*. 1258–1267. https://knowledge.amia.org/72332-amia-1.4602255/t003-1.4606204/t003-1.4606205/3417206-1.4606266/3415131-1.4606263

[27] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*. 328–337. https://doi.org/10.1109/ICSE-SEIP52600.2021.00042

[28] Ran Wang, Zhengyi Yang, Wenjie Zhang, and Xuemin Lin. 2020. An Empirical Study on Recent Graph Database Systems. In *Knowledge Science, Engineering and Management*. 328–340.

[29] Rui Yang, Yingying Zheng, Lei Tang, Wensheng Dou, Wei Wang, and Jun Wei. 2023. Randomized Differential Testing of RDF Stores. In *Proceedings of the 45th International Conference on Software Engineering: Demonstrations*. 136–140.

[30] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding And Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 283–294. https://doi.org/10.1145/1993498.1993532

[31] Michal Zalewski. 2022. American Fuzzy Lop (2.52b). https://lcamtuf.coredump.cx/afl/

[32] 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. https://doi.org/10.1145/3533767.3534409

[33] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970. https://doi.org/10.1145/3372297.3417260

Received 2023-02-16; accepted 2023-05-03